Flutter Framework

Desenvolva aplicações móveis
no Dart Side!

Casa do Código

LEONARDO H. MARINHO

ISBN

Impresso e PDF: 978-65-86110-26-5

EPUB: 978-65-86110-28-9

MOBI: 978-65-86110-27-2

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

PREFÁCIO

Programadores são máquinas de resolver problemas. Resolvemos problemas de vários segmentos e para vários tipos de pessoas, seja através de sistemas gerenciais, pontos de venda (pdv), gestão de educação, contabilidade ou saúde. Cada segmento necessita dos cuidados da tecnologia, e nós somos os artesãos que constroem estes artefatos de forma completamente manual através de um teclado e mouse. Assim como um oleiro que constrói vasos e artefatos com barro e argila, nós construímos softwares utilizando as ferramentas que melhor se adaptam aos projetos que vamos desempenhar. A escolha das ferramentas corretas impacta diretamente na qualidade do que estamos construindo quando produto final.

Neste livro, criaremos juntos aplicativos verdadeiramente performáticos, multiplataforma e atrativamente bonitos. Desconstruiremos a barreira que foi plantada na mente dos desenvolvedores para dispositivos móveis de que criar aplicativos é chato, custoso, lento e difícil. Estou escrevendo este livro para que seja um verdadeiro divisor de águas na sua carreira.

Flutter veio com uma proposta totalmente diferente de seus antecessores, o que traz bastante curiosidade aos desenvolvedores e esperanças para que seja um framework que realmente divida águas no meio tecnológico. A criação de aplicativos móveis principalmente para as plataformas Android e iOS é cada dia mais importante com o crescimento do público que dispõe de smartphones. Como especialistas em tecnologia precisamos nos atentar para este gigante mercado emergente e cheio de ótimas

oportunidades que clama por nós todos os dias.

Por isso, quero ser o seu aliado na caminhada com os estudos do Flutter e conseguir somar toda a minha experiência acadêmica e profissional para tornar isso possível e muito mais rápido. Conhecer a linguagem Dart e Flutter deixou de ser um extra e passou a ser uma exigência do mercado móvel.

Quer um bom incentivo para estudar Flutter? A Nubank, Agibank, iFood, e várias outras gigantes utilizam Flutter. Se elas já estão atentas à novidade, quem somos nós para ir na contramão?

Neste livro veremos quais dores o Flutter veio para curar, a arquitetura de software fantástica por baixo dele, razões pelas quais Dart foi a linguagem escolhida para o Flutter, o que são Widgets e como aproveitá-los ao máximo, gerenciamento de dependências, prototipação de aplicativos, requisições http, banco de dados local, testes automatizados, estilização de um aplicativo e muito mais! Tudo na prática, bem "*coding*".

O público-alvo deste livro são pessoas que tenham domínio de programação e que preferencialmente já tenham tido contato com programação para dispositivos móveis. Nada impede que novatos leiam e entendam os princípios apresentados aqui, porém, com conhecimento prévio o conteúdo será muito mais aproveitado.

Nos exemplos que serão apresentados utilizaremos o Flutter na versão 1.12.13+hotfix.8, e a linguagem Dart 2.7.0. Como normalmente saem muitas atualizações para o Flutter e a linguagem Dart, é bastante provável que no momento em que você estiver lendo este livro algo já tenha mudado de versão. Sem pânico, é possível rodar os exemplos sem problemas já que

normalmente o próprio Flutter acusa o que há de errado e dá o caminho para a alteração, quando ele próprio não a realiza sozinho.

Os códigos de todos os exemplos estão disponíveis para download no final de cada capítulo.

Boa leitura e sucesso!

DEDICATÓRIAS

A Deus, que me sustentou até aqui e aliviou o peso de todos os desafios gigantes que estou vivendo desde quando decidi entrar para o ramo tecnológico. A cada degrau que consegui subir na escada da vida, senti visivelmente o alívio e a poderosa mão Dele me guiando.

Aos meus pais, Avalcir e Adriana, o meu muito obrigado! Podem ter certeza de que nada foi em vão. Honrarei todo o esforço e dificuldades que passamos juntos para eu chegar até aqui. Vocês me trazem muito orgulho e faço de tudo para também orgulhá-los!

A Vanessa, minha eterna "irmãzinha mala" que meus pais me deram de presente, espero de coração poder sempre ajudá-la e aconselhá-la mostrando que os seus sonhos dependem apenas de você para se concretizarem! Desejo sucesso para você, e conte sempre comigo para a realização dos seus sonhos!

Aos amigos, ex-professores e pessoas maravilhosas que passaram/estão pela minha vida, muito obrigado pelos seus ensinamentos, puxões de orelha, conselhos e preocupação. Podem ter certeza de que nada disso foi em vão e que estou fundamentando a minha vida em tudo de melhor que vocês construíram no meu intelecto e caráter.

E por último, mas não menos importante, o meu muito obrigado aos membros da editora Casa do Código pela atenção, carinho e puxões de orelha para que eu conseguisse melhorar a minha escrita e entender os passos para a publicação deste livro.

AFINAL, QUEM É ESTE QUE VOS FALA?

Meu nome é Leonardo Herdy Marinho. Sou academicamente formado como analista e desenvolvedor de sistemas pela UNESA. Enquanto escrevo este livro, sou mestrando pela Universidade Federal do Rio de Janeiro (UFRJ), professor de Dart e Python na modalidade Ead e professor presencial de banco de dados.

Também tenho formação pela escola da vida em "engenharia de como funciona esse negócio?". Desde a minha infância busco entender os motivo e possíveis explicações sobre as coisas. Aos 13 anos quando ganhei o meu primeiro computador com acesso à internet, foi libertador! Daquele dia em diante a busca pelas respostas sobre a tecnologia se intensificou cada dia mais.

A primeira linguagem com que eu tive contato foi Delphi lá pelos meus 14 para 15 anos. Aprendi basicamente a copiar e colar código naquela IDE ultrassofisticada que parecia um genuíno painel de avião. Às vezes os códigos funcionavam, outras, não! Dependia da minha sorte já que eu não entendia a maioria das coisas que estavam acontecendo ali. Foi com 16 anos que tive acesso a um curso de linguagem C, presencial, e que realmente me ensinou a base da programação e me fez entender bastante sobre como eu poderia escrever um programa útil e que o meu computador entendesse exatamente como eu idealizei na mente.

Após o curso, consegui um estágio e logo em seguida entrei para a faculdade. Lá conheci pessoas e professores fantásticos que abriram portas de empregos e me apresentaram a tecnologia móvel

de que tanto tenho carinho até hoje. E lá se foi muito tempo trabalhando com mobile, e, não me arrependo nenhum instante de ter passado por essa estrada. Vi o Cordova Project nascer, jQuery mobile, Ionic Framework, AngularJS, React Native e muitas outras tecnologias que construíram o desenvolvimento móvel híbrido. Também assisti à nem tão recente mudança da Google do Java para o Kotlin e da Apple do Objective C para Swift como linguagem oficial para a criação de aplicativos. Propus-me realizar alguns desafios antes dos 30 anos, dentre os quais estava escrever um livro. Então, além de uma obra técnica o que você lerá daqui em diante é uma pequena parte de um grande sonho que estou realizando chamado vida unido a um amor profissional que é o desenvolvimento de aplicativos móveis que está tão em alta na última década.

Sumário

1 Introdução **1**

2 Quais dores o Flutter curou? **5**

3 O que preciso para desenvolver em Flutter? **10**

3.1 Criando Apps para iOS 10

3.2 Criando Apps para Android 11

3.3 Dica do autor 12

4 A arquitetura Flutter **14**

4.1 Estrutura de arquivos 14

4.2 Widgets 20

4.3 Componentes do Material Design 24

4.4 Scaffold 27

4.5 Stateful hot reload 35

5 A base do framework, Dart! **37**

5.1 A cereja do bolo 37

5.2 Entendendo a arquitetura Dart 39

5.3 A ponte que estamos derrubando usando AOT 43

5.4 Concorrência, divisão de tempo e recursos compartilhados 45
5.5 Interface unificada 47
5.6 Alocação de memória e cuidados com o lixo 49

6 Widgets, Widgets por toda a parte! 51
6.1 Stateless Widget 51
6.2 Stateful Widget 63
6.3 Mais um pouco de teoria sobre Widgets para explodir a sua mente 69

7 Dependências 71

8 Prototipação 83
8.1 Quem sabe para onde quer ir chega mais rápido 83
8.2 Cuidado com o Frankenstein 85

9 Avançando com o Flutter 89
9.1 HTTP 89
9.2 API 91
9.3 JSON 93
9.4 Costurando os retalhos 95

10 Criando um aplicativo complexo 98

11 Banco de dados 110
11.1 Um pouco sobre dados 110
11.2 Mão na massa! 115
11.3 Singleton 122
11.4 Nem só de providers vive um app 128

12 Testes automatizados em Widgets **145**

12.1 Para que testar? 145

12.2 Criando testes automatizados com Dart 147

13 Mudança do ícone do aplicativo **157**

14 Será o Flutter a bala de prata? **163**

14.1 A bala de prata 163

14.2 O Flutter é uma bala de prata? 164

14.3 Ionic + React Native + Flutter 166

14.4 E é aqui que a saga começa! 172

Capítulo 1

INTRODUÇÃO

No desenvolvimento de aplicativos para dispositivos móveis, há muitas particularidades que são bastante diferentes da criação de um site ou de aplicações desktop. Precisamos nos preocupar muito mais com a usabilidade, experiência, atratividade, interface gráfica e cores relevantes para a proposta do aplicativo. A otimização para reduzir ao máximo o consumo de hardware, requisições via internet e persistência local de dados também são outros grandes desafios.

Eu gosto de enxergar os aplicativos como um "trader" de informações. Eles basicamente são o meio de campo entre o servidor e o usuário. São a interface que envia ou recebe algum dado de uma base de dados e exibe de forma simples e elegante. É uma genuína obra de arte da abstração o que fazemos com esse tipo de tecnologia, que consegue em poucas telas e com campos simples exibir para o usuário tudo o que ele precisa saber de forma rápida, eficiente e bastante limpa. Bom, pelo menos assim deveria ser.

O desenvolvimento móvel, até então, vivia em uma gangorra. Para ganhar em performance, era necessário programar de modo nativo, ou seja, desenvolver "N" aplicativos cada um com a sua linguagem nativa para a respectiva plataforma. Por exemplo, criar

um aplicativo na linguagem Kotlin para o Android e outro, do zero, na linguagem Swift para o iOS. Mesmo sendo o "mesmo aplicativo" esse retrabalho era necessário. Por outro lado, se performance não fosse lá uma grande exigência, você poderia utilizar tecnologias como o Ionic, React Native, Native Script ou concorrentes similares. Esses são conhecidos como aplicativos híbridos: são em parte nativos e em parte tecnologias Web (HTML, CSS e JavaScript) ou outro tipo de linguagem.

As tecnologias híbridas até atenderam de forma bastante interessante o mercado por vários anos, mas os desenvolvedores sempre mantiveram guardada dentro de si a insatisfação na exibição de animações e do carregamento inicial quando o aplicativo era aberto. De fato é demasiadamente ruim saber que existem aplicações melhores e que você não consegue reproduzir de forma similar meramente por uma barreira da tecnologia empregada. Infelizmente, o cliente não entende essa parte e acaba nos culpando pela lentidão, mas, se não queremos criar dois aplicativos iguais com código nativo para ter ganho de desempenho, o que fazer?

Um mundo ideal seria poder obter o mesmo desempenho de processamento e renderização de uma aplicação nativa em uma aplicação híbrida que possibilitasse escrever apenas um código e executar para N plataformas. Com isso, dezenas ou centenas de milhares de reais seriam economizados na produção do código e a manutenção seria absurdamente facilitada por não haver redundância de software em linguagens diferentes. Também seria necessário bem menos mão de obra para criar a mesma coisa. São pontos positivos tanto para o cliente, que consequentemente pagará menos pelo mesmo, para o empreendedor, que poderá

reduzir custos e maximizar lucros, como para a equipe de desenvolvimento, que precisará se esforçar menos para entregar algo funcional e agradável.

É por isso que, neste livro, venho apresentar para você o que eu chamo de "framework do mundo ideal", ou, **Flutter**. O time de engenheiros que o Google levantou para se responsabilizar pela criação da tecnologia, assim como na evolução da linguagem Dart, deu um verdadeiro show de competência. Temos praticamente toda a performance e agilidade de um aplicativo nativo em algo "híbrido" e de fácil compreensão. E é de rápida aprendizagem: com no máximo uma semana você já estará criando seus primeiros aplicativos úteis. Caso você tenha experiência com aplicações híbridas que utilizam *web views* para renderizar HTML, CSS e JavaScript, com certeza você sentirá na pele a diferença na velocidade e fluidez das aplicações Flutter, tanto em tempo de compilação quanto no carregamento e uso.

O Flutter é um framework que implementa o modelo de programação reativa inspirado nos moldes do React. A ideia é que ele, junto com a linguagem Dart, seja a principal forma de criar aplicativos móveis para o próximo sistema operacional que será lançado pelo Google para suceder o Android. Tudo no Flutter é organizado como **Widget**, independente de ser um botão, campo de texto ou espaçamento. Basicamente, os Widgets são interpretados de forma que viram componentes nativos para o Android e para o iOS assim que o código é executado, então, o produto final não é algo em outra linguagem se comunicando com a plataforma nativa por meio de uma ponte (como o Ionic e simulares fazem), mas sim, código nativo de alto desempenho.

Nos próximos capítulos vamos viajar pelas funcionalidades desse framework fantástico e também criar alguns aplicativos bastante legais partindo do clássico “hello world” até a integração com API, uso de banco de dados locais e afins. Veremos melhor o conceito de Widget e quais são os mais comuns utilizados cotidianamente. Nossos amigos do Google já prepararam a maior parte do caminho, agora, nos resta aproveitar e criar aplicativos bonitos, performáticos e com uma única base de código. E aí, topa o desafio?

Capítulo 2

QUAIS DORES O FLUTTER CUROU?

1. Compilação absurdamente mais rápida que a do Ionic e de outras tecnologias similares.

2. O *hot reload* (se não sabe do que se trata, mais para a frente veremos em detalhes) faz os desenvolvedores ganharem praticamente um minuto a cada alteração que precisarem realizar e visualizar o resultado. Outra coisa muito legal que ele trouxe foi a não mudança do estado da aplicação. Isso economizou bastante tempo pós compilação, principalmente para o desenvolvimento de formulários em que precisamos preencher todos os campos repetidamente a cada mudança realizada no código. Assim, ele mantém o estado da aplicação, simplesmente, como se nada tivesse ocorrido e isso torna o hot reload muito fantástico e satisfatório. Adeus a preencher os mesmos campos dezenas de vezes por dia!

3. Tempo de boot (abertura da aplicação) muito mais rápido e profissional do que todas as tecnologias não nativas com as quais tive contato.

4. Fluidez independente do que tem na tela, seja um botão ou um vídeo. Em diversos testes (meus e de alguns amigos

desenvolvedores junto aos seus usuários), nenhum aplicativo em Flutter travou até o momento de escrita deste livro. Realizei os testes em aparelhos Android e iOS partindo do melhor hardware possível até o pior (incluindo sistemas antigos e desatualizados). Isso nos surpreendeu bastante e mostrou que a arquitetura do framework e do Dart é realmente bastante sólida e concisa. Não se trata de mais um framework mediano, e sim, de algo realmente inteligente e bem feito.

5. Beleza! Conseguimos criar coisas bem bonitas com o Flutter. Não somos designers (ou, geralmente não), no máximo brincamos um pouquinho com edição de imagem e criação de "memes" nas horas vagas para brincar com os amigos. Construir um lindo layout não é a grande vocação dos desenvolvedores, e a interface de usuário das tecnologias anteriores não ajudava em praticamente NADA com isso. Nunca ficamos satisfeitos com o que criamos, mesmo quando agrada o cliente. O Flutter, sem esforço, traz Widgets que são uma obra de arte fenomenal!

6. A forma como o código é distribuído também é simplesmente fenomenal. A linguagem Dart passa a impressão de que foi criada sob encomenda para o Flutter. Quem vem de linguagens como C, C++, C#, Java e similares sente uma facilidade absurda e a curva de aprendizado se torna quase instantânea. Para quem vem de tecnologias Web, como eu, basta um pouquinho de dedicação depois do horário de trabalho (uma hora por dia no máximo) para em pouco tempo você começar a criar algo interessante. Flutter e Dart são um grande casamento do qual somos todos padrinhos! Quer contribuir com a "gravata" da festa? Existe um repositório com o projeto Flutter totalmente aberto para a comunidade participar da evolução do framework, basta acessar

a documentação no site do Flutter e se informar melhor. Você também pode criar pacotes para disponibilizar no repositório de extensões da linguagem.

7. Velocidade de carregamento e fluidez das transições de tela e demais efeitos. A leveza que conseguiram criar no Flutter é realmente notável e merece a nossa atenção. Ouso dizer que alguns aplicativos criados com o framework executam transições mais rápidas que outros criados de forma nativa. Não que o Flutter seja mais rápido que o nativo, mas sim pela otimização que o time de desenvolvimento se preocupou em criar, contra a não preocupação dos desenvolvedores nativos por acharem que qualquer código será performático exclusivamente por estar em Java/Kotlin ou Objective C/Swift. A fluidez nos permite viajar na imaginação e criar animações das mais variadas tornando a usabilidade muito divertida.

8. Exclusão de potenciais problemas para a comunicação com o hardware. Aplicativos híbridos sofrem até hoje para se comunicarem com os recursos de hardware sejam eles câmera, giroscópio, acelerômetro, GPS ou outros. Para tal comunicação eles precisam de extensões e plugins criados pela comunidade que, na maioria das vezes, são altamente problemáticos por precisarem transportar informações de uma linguagem para a outra por meio de APIs internas de comunicação com o código nativo. A troca de contexto sempre é muito custosa. Um bom exemplo técnico trazendo para a realidade seria algo como:

- O usuário aperta o ícone da câmera em uma aplicação híbrida.
- O código do aplicativo híbrido (JavaScript) comunica-se

com a web view (navegador) que o executa solicitando a abertura da câmera.
- A web view informa ao Java (no caso do Android) ou Swift (no caso do iOS) da parte nativa do projeto que necessita da câmera.
- O Java/Swift por sua vez, solicita ao sistema operacional dizendo que o aplicativo precisa abrir a câmera.
- O sistema operacional abre a câmera e responde ao Java/Swift que a câmera está aberta.
- O Java/Swift repassa para a web view a informação sobre a abertura da câmera.
- A web view repassa a mensagem sobre a câmera para o JavaScript.

Isso não é lá das coisas mais performáticas e livres de falhas. Com a remoção de diversas camadas de burocracia, podemos praticamente nos comunicar com o hardware do dispositivo usando Dart através de uma fina camada de C e C++ lidando com a linguagem nativa. E só! Não tem mais nada entre o seu aplicativo e o sistema operacional. Não é fantástico?

9. Segurança no acesso às funcionalidades. Quem vem do JavaScript sabe que de seguro ele não tem nada (refiro-me ao uso dele principalmente no front-end). Qualquer erro simples pode atrapalhar a execução de toda uma cadeia de código crítica acarretando no não funcionamento correto de alguma função que controla as permissões de acesso a uma tela, por exemplo. Grande parte dos desenvolvedores não sentia segurança na criação de aplicativos híbridos quando algo exigia controle de permissões nas telas, afinal, algum erro na consulta do banco de dados local, ou em uma requisição HTTP, poderia acarretar na quebra por

completo do JavaScript e consequentemente liberar funcionalidades que não deveriam ser liberadas. Graças à estrutura interna de funcionamento do Dart e do Flutter, você pode ficar totalmente em paz quanto a esse tipo de problema e não necessita ter insegurança ao criar módulos de permissão de acesso. Eles funcionarão independente de haver algum erro ou não em outras funcionalidades que não estão diretamente ligadas a ele.

10. Simplicidade em configurar o ambiente de desenvolvimento. Pode ser facilmente configurado no Windows, Mac ou Linux em questão de minutos sem nenhuma burocracia. Configurar variáveis de ambiente, preocupar-se em instalar a versão específica do Java, nunca mais atualizar para não desconfigurar tudo, e ter que adivinhar o que houve quando o ambiente parar de funcionar, isso está definitivamente fora do mundo Flutter. Além de muito simples para instalar e configurar, o comando `flutter doctor` analisa todo o ambiente de desenvolvimento instalado e nos dá um relatório quase médico fazendo jus ao seu nome. Caso qualquer coisa estiver errada, ele instantaneamente alertará. Quando configurávamos ambientes Ionic ou NativeScript ou React Native na empresa, dependendo do sistema operacional ou das coisas instaladas, levávamos pelo menos um dia inteiro para fazer funcionar plenamente com todos os erros corrigidos. Complexidade desnecessária que agora não precisamos mais em nossas vidas! Paralela à facilidade do próprio Flutter, a comunidade tem desenvolvido soluções para realizar a instalação e configuração automática do ambiente como o Atelo, por exemplo. Para saber mais sobre ele acesse https://atelo.unicobit.com/.

Capítulo 3

O QUE PRECISO PARA DESENVOLVER EM FLUTTER?

3.1 CRIANDO APPS PARA IOS

Por mais que seja doloroso, preciso lhe contar isto. Você **NECESSITA** ter um Mac (MacBook/MacMini/iMac) se quiser criar aplicativos para iOS. Não importa qual, escolha o que melhor cabe no seu orçamento e realidade. Recomendo fortemente o Macbook Pro, existem várias opções usadas excelentes caso você não possa arcar com um novo. Esqueça o hackintosh, máquinas virtuais e quaisquer outros jeitinhos. Artifícios dão muita dor de cabeça e perda de tempo. Se quer desenvolver softwares de qualidade invista no equipamento necessário - o retorno financeiro virá, pode ter certeza. A partir de agora, se você quiser desenvolver profissionalmente para a Apple precisará de equipamento profissional Apple, sem choro! Garanto que não vai se arrepender, nunca me arrependi do meu.

Se você pensa que os investimentos acabaram, calma que tem mais. Para compilar as aplicações para o iOS e enviar para a AppStore obrigatoriamente você precisa ter um device Apple

conectado ao seu Mac, logo, você também precisará de um iPhone ou iPad. Pronto, já pode voltar a respirar. A Apple é bastante exigente no desenvolvimento de aplicativos e nos força ao máximo a utilizarmos os hardwares deles, mas, no final quando você vê a qualidade e estabilidade do software, é simples de entender a razão pela qual fazem isso. Não recebo pela publicidade, estou apenas compartilhando as minhas boas experiências.

Da parte de software para desenvolvimento, basta você instalar o aplicativo XCode que está disponível na AppStore e pronto! Tudo o que precisaremos para os ciclos de desenvolvimento está preparado.

Como fazemos para publicar um aplicativo na AppStore? Bom, primeiro precisaremos de uma conta de desenvolvedor. Para criar aplicativos e testar, você não precisa da conta premium da Apple. Com um simples cadastro no programa de desenvolvedores você terá o que precisa para trabalhar, mas, para publicar e manter o suporte online é necessário, sim, desembolsar esse dinheiro que no momento que escrevo este livro gira em torno de 99 dólares anuais. O mundo Apple é bem legal, mas exige investimento.

3.2 CRIANDO APPS PARA ANDROID

Tudo o que o iOS tem de complexo o Android tem de simples no que diz respeito ao desenvolvimento e publicação. Não precisamos de um computador específico, o desenvolvimento de aplicativos Android pode ser feito nos três maiores sistemas operacionais da atualidade: Windows, Linux ou macOS.

É bom ter um dispositivo Android para testar fielmente as

alterações e garantir o bom funcionamento da aplicação. O emulador em alguns casos resolve o seu problema também, mas, nada simulado é tão real quanto o original. Tenha um aparelho Android em mãos para ter garantia de qualidade e testes fidedignos.

Para publicar os aplicativos na Google Play você também precisará de uma conta de desenvolvedor assim como a Apple exige, porém, a taxa não é anual. Basta pagar uma vez o valor que no momento que escrevo este livro é de 25 dólares e pronto. A cobrança é apenas para validar que você realmente é alguém interessado e sério e não um robô querendo fazer bagunça na loja.

Preciso ter a conta paga de desenvolvedor Google para apenas criar meus aplicativos e testar? Não! Você só precisa pagar a conta caso queira publicar na loja de aplicativos e disponibilizar para download.

3.3 DICA DO AUTOR

O meu equipamento de trabalho se resume a um iPhone mais antigo (não falarei a versão para não passar vergonha), mas funciona perfeitamente! Um celular com o Android mais recente e um Macbook Pro. Esses equipamentos me servem perfeitamente e estão me atendendo há alguns anos. Tenho certeza de que também atenderão a você. Caso necessite testar os aplicativos em uma versão específica do iOS ou Android, não precisa se desesperar e sair comprando a loja de celulares toda - é aí que entram os emuladores para nos ajudar. Não recomendo utilizá-los durante todo o desenvolvimento devido ao alto consumo de memória e processamento do computador, mas para testes pontuais são de

suma importância. Também é interessante manter o sistema do dispositivo Apple e do macOS devidamente atualizados. Essa é uma exigência deles, normalmente quando trabalhamos com versões de anos anteriores várias coisas vão deixando de ganhar suporte e simplesmente param de funcionar.

É válido ressaltar que não estou dizendo para você fazer o mesmo que eu, sinta-se livre para construir o seu ambiente de trabalho como quiser. Essas são dicas apenas para facilitar a vida de quem está começando agora e não tem um norte a respeito de qual equipamento é necessário para iniciar os estudos e trabalho. Caso queira desenvolver por hora apenas para Android, sinta-se livre para utilizar qualquer outro notebook Windows ou Linux.

Capítulo 4

A ARQUITETURA FLUTTER

4.1 ESTRUTURA DE ARQUIVOS

O Flutter dispõe de uma estrutura de pastas e arquivos que já vem por padrão quando criamos um novo projeto. Precisamos dominar esses arquivos como a palma de nossas mãos para ter um desenvolvimento ágil e conciso. A seguir temos uma imagem com a estrutura de um projeto Flutter:

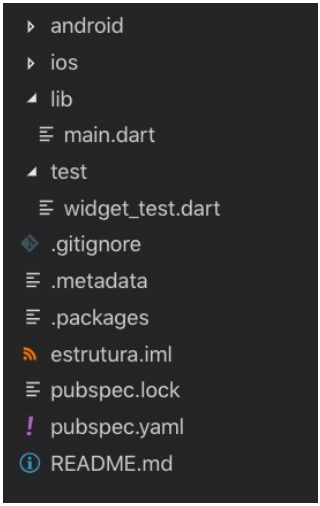

Figura 4.1: Estrutura do projeto

android: nessa pasta, temos o projeto Android nativo, criado normalmente em Java. Quando o código Flutter for executado, ele automaticamente é compilado para Java e executado a partir dessa pasta.

Apesar de ser bom ter conhecimento na linguagem Java e Kotlin, não é obrigatório. Não precisamos obrigatoriamente conhecer as linguagens nativas, afinal, o Flutter está aí para isso! A seguir temos uma imagem do conteúdo da pasta `android`:

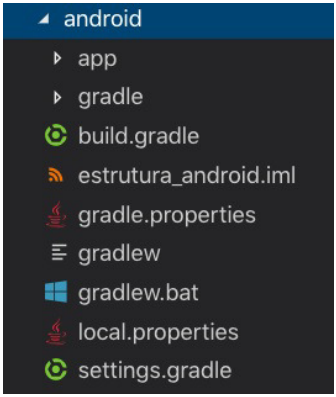

Figura 4.2: Pasta android

iOS: nessa pasta, temos o projeto iOS nativo, criado em Swift. Quando o código Flutter for executado ele automaticamente é compilado para esta linguagem e executado a partir da pasta `iOS`.

Assim como foi citado a respeito do Java e Kotlin, não é obrigatório ter conhecimento do Swift para criar seus aplicativos Flutter. A seguir temos uma imagem do conteúdo da pasta `iOS`:

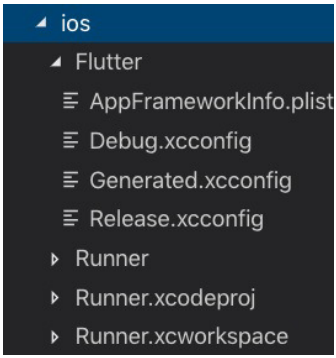

Figura 4.3: Pasta iOS

lib: nessa pasta, temos por padrão o arquivo `main.dart`, nele está o código que inicia a nossa aplicação. Dentro da pasta `lib` podemos criar outras pastas e organizar os arquivos escritos em Dart da forma que acharmos mais organizada e justa para o correto funcionamento do aplicativo.

Você pode brincar à vontade dentro dessa pasta, só não pode substituir o nome do arquivo `main` nem remover o método `main` de dentro dele, se não o aplicativo não executará. A seguir temos uma imagem do conteúdo do arquivo `main.dart`:

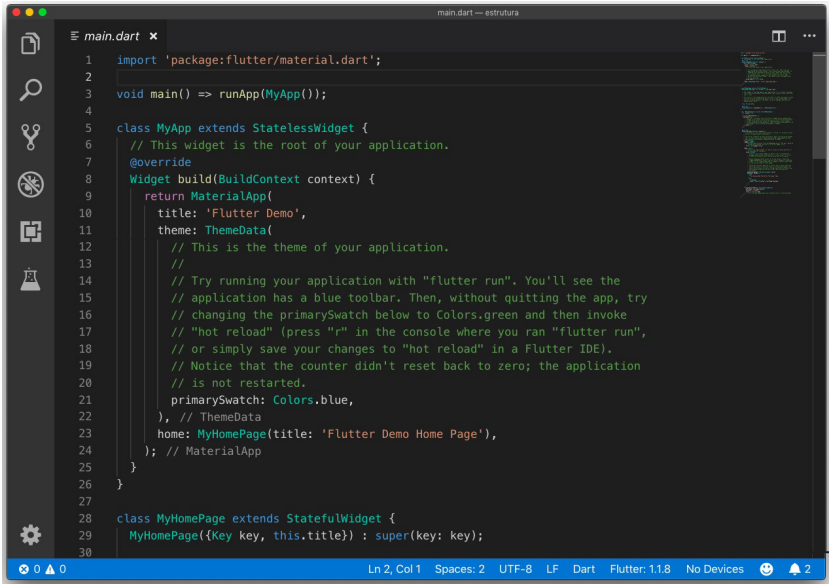
main.dart — estrutura
main.dart
1 import 'package:flutter/material.dart';
2
3 void main() => runApp(MyApp());
4
5 class MyApp extends StatelessWidget {
6 // This widget is the root of your application.
7 @override
8 Widget build(BuildContext context) {
9 return MaterialApp(
10 title: 'Flutter Demo',
11 theme: ThemeData(
12 // This is the theme of your application.
13 //
14 // Try running your application with "flutter run". You'll see the
15 // application has a blue toolbar. Then, without quitting the app, try
16 // changing the primarySwatch below to Colors.green and then invoke
17 // "hot reload" (press "r" in the console where you ran "flutter run",
18 // or simply save your changes to "hot reload" in a Flutter IDE).
19 // Notice that the counter didn't reset back to zero; the application
20 // is not restarted.
21 primarySwatch: Colors.blue,
22), // ThemeData
23 home: MyHomePage(title: 'Flutter Demo Home Page'),
24); // MaterialApp
25 }
26 }
27
28 class MyHomePage extends StatefulWidget {
29 MyHomePage({Key key, this.title}) : super(key: key);
30
0 0
Ln 2, Col 1 Spaces: 2 UTF-8 LF Dart Flutter: 1.1.8 No Devices 2

Figura 4.4: Arquivo main

test: nessa pasta, temos por padrão o arquivo `Widget_test.dart` , onde está o código desenvolvido especialmente para testar o funcionamento dos Widgets e interação com o usuário. A aplicação default que temos quando criamos um projeto Flutter já vem com os testes configurados para ela. Os testes vindos de presente no aplicativo padrão verificam se os Widgets especificados existem, se o contador está zerado e se o botão flutuante vai acrescer mais 1 no valor exibido na tela. A seguir temos uma imagem do conteúdo do arquivo `Widget_test.dart` :

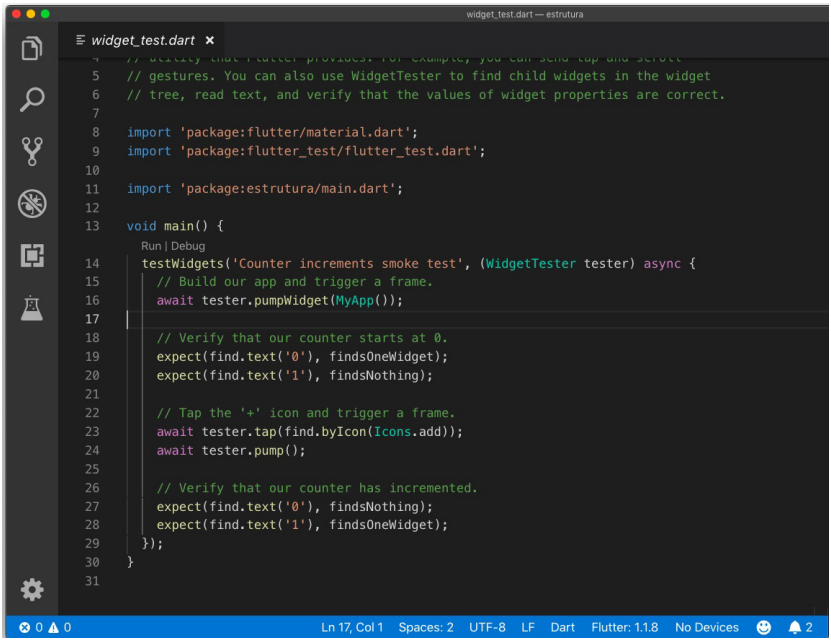

Figura 4.5: Arquivo de testes

.gitignore: é nesse arquivo que nos comunicamos com o controle de versão, o que especificarmos no `gitignore` não será versionado. Não precisamos salvar no nosso repositório de códigos as pastas com as dependências que podem ser tranquilamente baixadas posteriormente, assim, economizamos espaço no disco do servidor responsável pelo repositório e tornamos muito mais rápido o envio e recebimento dos arquivos. Com o `gitignore` gerado pelo Flutter apenas os códigos Dart e coisas importantes do Flutter são salvos no repositório; arquivos de compilação, temporários e afins não entram no bolo.

.metadata: o arquivo de metadados mantém algumas propriedades específicas do nosso projeto Flutter, e ele é

responsável por fornecer os dados para atualizações do framework, extensões e similares. É bastante importante versionar este arquivo (ou seja, não o colocar no `gitignore`) e principalmente não mexer em seu conteúdo, afinal, ele é autogerado e administrado pelo SDK Flutter.

.packages: é aqui que o SDK Flutter salva as urls das dependências que ele necessita mais frequentemente. Basicamente é um arquivo que indexa todos os arquivos principais de funcionamento do Flutter para caso houver a necessidade de executá-los não termos problemas com performance e demora no tempo de execução.

pubspec.yaml: podemos dizer que o arquivo `pubspec` é o coração das dependências e controle dos nossos aplicativos. Nele especificamos quais dependências/extensões queremos utilizar (http, lazy load, bloc), o nome do projeto e descrição, versão do SDK Flutter que queremos utilizar, o diretório dos nossos arquivos assets, isto é, nossos arquivos de fontes, imagens, vídeos e afins. Uma dica de ouro para se dar muito bem com esse arquivo é ler todos os comentários que vêm por padrão nele explicando cada funcionalidade do arquivo.

pubspec.lock: é efetivamente o arquivo que o Flutter lerá. Basicamente, ele é um compilado do `pubspec.yaml` só que de forma otimizada para a fácil leitura pelo SDK. O arquivo `lock` é gerado na primeira vez que você cria o projeto, e só é atualizado caso você o exclua e execute novamente a compilação. Ele é responsável por manter a compatibilidade das dependências.

README.md: como a extensão do próprio já diz, ele é escrito utilizando a linguagem de marcação Markdown. Este arquivo não

é essencial para o funcionamento do projeto, é apenas um descritivo para você criar anotações sobre o aplicativo e dicas como as de instalação, por exemplo, caso outras pessoas possam baixar o projeto. Sinta-se livre para anotar o que achar pertinente neste arquivo, ou, até mesmo removê-lo.

estrutura.iml: se você não criou o seu projeto com o nome estrutura com certeza esse arquivo seu terá o nome do seu projeto com a extensão `iml` . Basicamente é um facilitador para o Dart se comunicar com o interpretador Java na hora de gerar o aplicativo para o android. Algumas versões do Flutter simplesmente não geram este arquivo. Caso o seu projeto não tenha, não se preocupe. Assim como o arquivo metadata, ele é autogerado e não deve ser editado manualmente.

4.2 WIDGETS

Os Widgets são o que você vai utilizar para construir o seu aplicativo. Uma lista é um Widget, um botão é um Widget, uma barra de busca é um Widget e todo e qualquer outro tipo de elemento que você possa precisar para criar a interface gráfica do aplicativo será um Widget. O Flutter dispõe de duas modalidades básicas de Widgets: os personalizados e os básicos. Os Widgets básicos são os que já vêm por padrão no SDK Flutter (botões, campos de texto, ícones, listas...) e os personalizados são os que você pode criar de acordo com a sua necessidade, ou, baixar de algum outro desenvolvedor que já passou pela mesma necessidade que você e o criou.

Os Widgets do Flutter trabalham em cima do padrão de programação reativa inspirada nos moldes do React. Basicamente,

eles descrevem como a sua interface gráfica e estados da aplicação devem funcionar. Quando o estado de um Widget muda, ele recria a sua "existência", mudando exatamente o que foi alterado, realizando alterações mínimas na árvore de componentes da página. Com isso, não precisamos atualizar a tela inteira para mudar o valor de texto de um campo de nome, por exemplo. Apenas aquele componente em particular é recriado através da compilação `JIT` (modo de desenvolvimento) com a alteração específica.

Como criar um aplicativo simples

Vamos criar um aplicativo extremamente básico para começar a entender melhor os Widgets? Não é preciso ensinar como instalar o SDK do Dart e do Flutter e editores de texto ou IDEs. Essa explicação é facilmente encontrável na documentação oficial do framework https://www.dartlang.org/guides/get-started/, e já temos excelentes tutoriais pela internet com o passo a passo detalhado para instalar e configurar o ambiente. Vale falar novamente do instalador automático Atelo para ocasiões que você não sinta a necessidade de configurar o ambiente de forma manual. Para saber mais sobre ele, acesse https://atelo.unicobit.com/.

A partir de agora vou iniciar do pressuposto de que você já configurou tudo certinho e, acabou de criar um projeto default do Flutter, ok? Abra o arquivo `main.dart` no seu editor de texto ou IDE, apague tudo o que há dentro dele, e digite o seguinte trecho de código:

```
import 'package:flutter/material.dart';
```

```
void main() {
  runApp(
    Center(
      child: Text(
        'Hello, world!',
        textDirection: TextDirection.ltr,
        style: TextStyle(fontSize: 30.0),
      ),
    ),
  );
}
```

O resultado que você deverá encontrar ao executar este código precisa ser exatamente o da imagem a seguir:

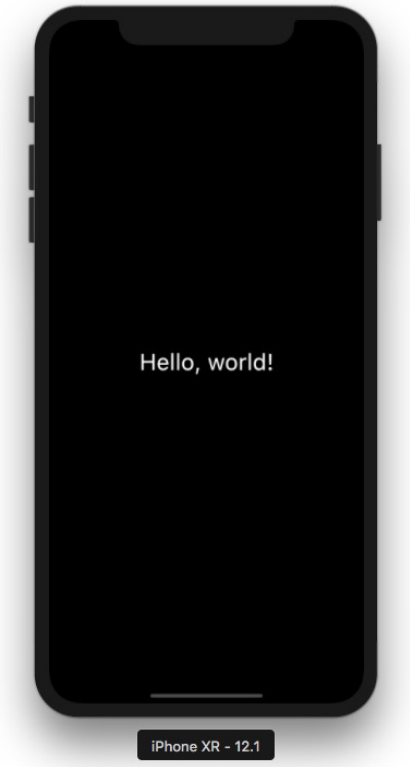

Figura 4.6: Hello world

Vamos entender o que acabamos de fazer. Na linha 1, encontramos um `import` chamando o arquivo `material.dart`. Este arquivo contém toda a biblioteca de Widgets necessários para desenvolver os seus aplicativos utilizando o padrão `Material Design` (Android). Sem essa linha, nada feito! O Flutter não saberia o que é um "Center", por exemplo. Então, obrigatoriamente precisamos ter um `import` que disponibilize os Widgets essenciais.

Na linha 3, temos a função `main`, que é o centro da execução de um aplicativo em Dart (se assemelha bastante ao C ou Java). O início do código é dado dentro dessa função. Na linha 4 temos a função `runApp`, que pega o Widget fornecido e faz dele a raiz da árvore de Widgets. Neste exemplo, a árvore de Widgets consiste em dois Widgets, o Widget `Center` e seu filho, o Widget `Text`. A estrutura força o Widget raiz a cobrir a tela, o que significa que o texto "Olá, mundo" termina centrado. A direção do texto também precisa ser especificada, no nosso exemplo, informamos que a direção do texto é a `ltr`, ou seja, "left to right" (da esquerda para a direita). Também tomamos a liberdade de inserir a propriedade style no nosso Widget `Text` para especificar que o tamanho da fonte será de 30 pixels.

Caso queira ver este projeto na íntegra, o código está disponível no seguinte endereço:

https://github.com/leomhl/flutterbook_hello_world/

Vale ressaltar que alguns Widgets esperam outro Widget filho, ou "child", e outros esperam um objeto array de filhos, ou "children". No final das contas, tudo é uma grande árvore de Widgets e seus filhos.

4.3 COMPONENTES DO MATERIAL DESIGN

Por padrão, o Flutter traz vários componentes gráficos prontos por meio da biblioteca `material dart` . Com esses componentes, podemos seguir fielmente o modelo `Material Design` (modelo de componentes e estilização criado e especificado pelo Google para a construção de interfaces gráficas). Para utilizar essa biblioteca, precisamos desenvolver o nosso aplicativo dentro de uma chamada do `MaterialApp` , como no exemplo:

```
void main() {
  runApp(MaterialApp(
    title: 'Flutter Framework',
    home: HomePage(),
  ));
}
```

Um aplicativo que utiliza o `material dart` é iniciado com o Widget `MaterialApp` , que cria vários Widgets úteis na raiz do aplicativo, incluindo um Navegador que gerencia uma sequência de Widgets identificados por sequências hierárquicas, também conhecidos como "rotas".

O Navegador permite fazer a transição sem problemas entre as telas do seu aplicativo no modelo de pilha. Cada nova página aberta vai para o topo da pilha. Basicamente, a última página a abrir é a primeira a ser fechada.

Usar o Widget `MaterialApp` é opcional, mas é uma boa prática que traz diversas vantagens e, principalmente, facilidades. Acompanhe um aplicativo criado utilizando o `Material Design` (não se prenda ao que você não entende ainda, nas próximas páginas explicarei mais detalhadamente tudo isso).

```
import 'package:flutter/material.dart';
```

```
void main() {
  runApp(MaterialApp(
    title: 'Flutter Tutorial',
    home: TutorialHome(),
  ));
}

class TutorialHome extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        leading: IconButton(
          icon: Icon(Icons.menu),
          tooltip: 'Menu de navegação',
          onPressed: null,
        ),
        title: Text('Exemplo de título'),
        actions: <Widget>[
          IconButton(
            icon: Icon(Icons.search),
            tooltip: 'Buscar',
            onPressed: null,
          ),
        ],
      ),
      body: Center(
        child: Text('Olá mundo!'),
      ),
      floatingActionButton: FloatingActionButton(
        tooltip: 'Adicionar',
        child: Icon(Icons.add),
        onPressed: null,
      ),
    );
  }
}
```

O resultado esperado para este código é:

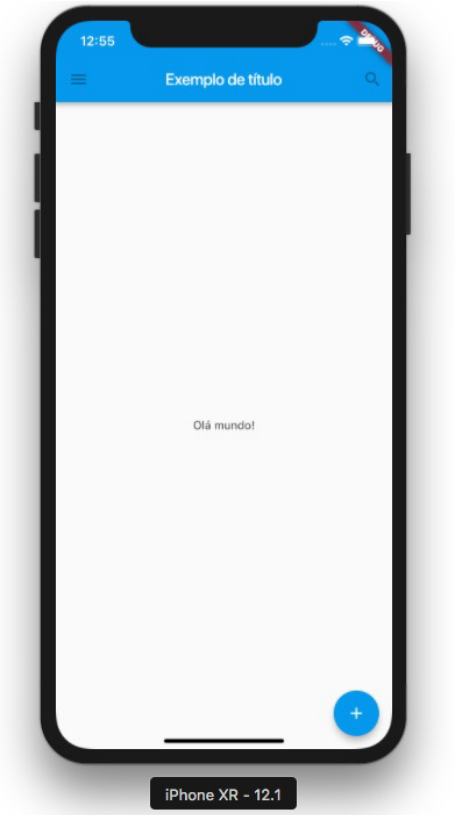

Figura 4.7: Hello world botões

O elemento `Scaffold` utilizado no nosso código implementa a estrutura do layout básica do `Material Design` . É por aqui que você deve começar a trabalhar para utilizar o Material nos seus projetos. É através dele que podemos especificar uma *toolbar*, configurar o texto, ícones e alinhamentos dela. Também especificamos qual é o corpo da página, ou, *body*. Podemos ainda especificar botões flutuantes, cor do background da página, botões de navegação e monitorar alterações de estado na aplicação.

4.4 SCAFFOLD

Para utilizar o `Scaffold` é necessária a implementação do `AppBar` e do `Body`. São o mínimo que precisamos para criar o essencial do `Material Design`. Sem essas propriedades, não podemos considerar necessariamente que um aplicativo é implementado em cima do `Material Design`.

Vamos acompanhar agora um exemplo bastante customizado do uso do Scaffold para entendermos até que ponto podemos chegar com ele. Claro que ainda não criaremos códigos de alta complexidade, a ideia aqui é entender tudo partindo bem do início.

O resultado que buscamos obter é o da imagem a seguir:

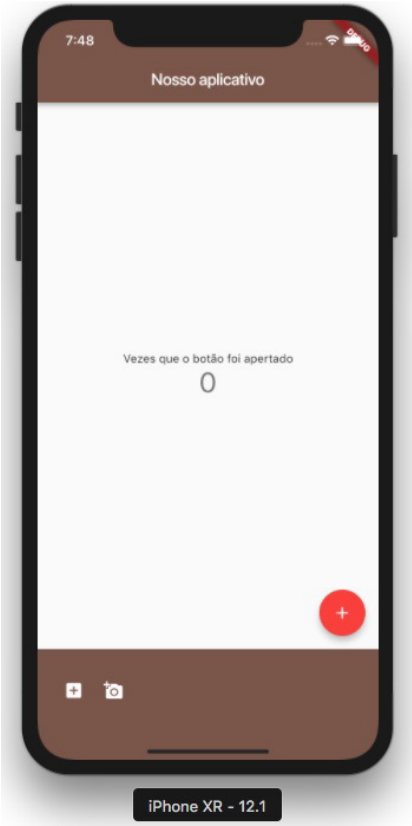

Figura 4.8: Exemplo de uso do Scaffold

É claro que na prática não criaremos um layout feio em que nada combina com nada como este, mas foi proposital. Tanto a `toolbar` quanto a `footerbar` estão em marrom, o `float button` em vermelho e o restante em preto/cinza e branco. Estou descrevendo as cores caso você esteja lendo uma edição em preto e branco do livro. Cada vez que pressionamos o botão flutuante (o circular com o ícone de mais), o contador adiciona mais 1 na contagem e, cada vez que pressionamos algum dos botões da nossa

barra de rodapé, ele dará um print no console de debug do nosso editor de texto/IDE. Legal, não? Mas como faz para criar? Não é difícil, mas isso exige algumas coisas que ainda não conhecemos. Por isso, vamos pontuar trecho por trecho.

Finalmente, show me the code!

Daqui para a frente, o universo Flutter está prestes a expandir e explodir dentro das nossas mentes. Um verdadeiro Big Bang computacional. A partir deste ponto começaremos a entender verdadeiramente a estrutura do projeto e como o Flutter realmente funciona. Até aqui, pegamos leve. De agora em diante, a brincadeira vai começar a ficar realmente divertida! Primeiro, começamos com os imports e a criação da classe que a chamada `main` vai utilizar.

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {

  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Livro sobre Flutter',
      home: MyHomePage(),
    );
  }
}
```

Na primeira linha do trecho de código, estamos importando o `material.dart` para utilizarmos os Widgets incríveis que ganhamos de presente do time que criou o Flutter. Na linha seguinte, estamos chamando o método `main` utilizando uma `arrow function`, que é uma maneira simplificada para chamar outra função. A `arrow function` aponta para a função

`runApp()` que carrega a nossa classe `MyApp`. Então, a linha 2 basicamente executa dentro da função `main` a função `runApp` passando como parâmetro o construtor da classe `MyApp`.

Já a classe `MyApp` estende uma classe que o SDK Flutter proporciona chamada `StatelessWidget`. Teremos um capítulo apenas para explicar a parte `Stateless` e `Statefull` dos Widgets, ok? Não se prenda a essa pequena parte agora. A classe `MyApp` conta com um método chamado `build` que retorna o Widget do `Material Design`. É o método `build` que constrói toda a nossa aplicação, ele é um dos mais importantes, perdendo apenas para o método `main`. O Widget `MaterialApp` recebe um título para a aplicação e um construtor da tela inicial, ou seja, a `HomePage`.

Nosso próximo passo é fazer o estado que será utilizado pela nossa classe. Para isso, vamos criar um `StatefulWidget`.

```
class MyHomePage extends StatefulWidget {
  _MyHomePageState createState() => _MyHomePageState();
}
```

A classe `MyHomePage` é a ponte entre a classe principal construtora do nosso aplicativo e o gerenciamento de estados para podermos brincar com variáveis, campos de texto com valores que o usuário pode alterar e similares.

Criada a ponte com o estado, precisamos criar o próprio. É dentro do `_MyHomePageState` que tudo o que é mutável na nossa classe se localiza. Como criaremos uma página com um contador, precisamos de um método que incrementa uma variável sempre que for chamado.

```
class _MyHomePageState extends State<MyHomePage> {
```

```
int _counter = 0;

void _incrementCounter() {
  setState(() {
    _counter++;
  });
}
```

A classe `_MyHomePageState` estende a classe `State` do tipo `MyHomePage`, ou seja, é uma extensão de classe aos moldes da nossa classe `MyHomePage`. Apesar de parecer mirabolante, com o tempo isso vai fazer muito sentido. É toda essa tipagem bastante específica que torna a linguagem Dart tão performática e perfeita para o que estamos criando aqui! Logo abaixo, temos declarada uma variável chamada `_counter` que recebe o valor 0.

No Dart o *underline* é utilizado para tornar algo privado dentro de uma classe ou método, então tudo o que estamos vendo que é declarado com `_` antes do nome é privado. O método `_incrementCounter()` não retorna nada, portanto, na declaração vemos um `void` e, dentro dele, temos uma chamada de `setState` recebendo uma função anônima que atribui +1 na variável `_counter`. É esta variável que está encarregada de contar quantas vezes o usuário do aplicativo apertou o botão flutuante (`float action button`).

A função `setState` é o que avisa ao Flutter de que algo foi modificado e que ele precisa realizar uma análise dos componentes da tela e renderizar novamente o que foi alterado. Basicamente é o `setState` que informa que houve alguma mudança na exibição para o usuário. Sem ele, nada feito! Por mais que uma variável mude não será atualizado no componente visual caso não o tenhamos.

Com a parte que controla o contador feita, precisamos criar os Widgets que serão exibidos para o usuário, para isso trabalharemos com o `Scaffold` e especificaremos como será a `AppBar` e o corpo da página.

```
@override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(
       title: Text('Nosso aplicativo'),
       backgroundColor: Colors.brown,
     ),
     body: Center(
       child: Column(
         mainAxisAlignment: MainAxisAlignment.center,
         children: <Widget>[
           Text('Vezes que o botão foi apertado',),
           Text('$_counter', style: Theme.of(context).textTheme.
display1,),
         ],
       ),
     ),
```

A annotation `@override` nos indica que o método logo abaixo dela está sendo sobrescrito, ou seja, a classe estendida (`State`) contém o método `build` . Mas, em modo padrão, ele não faz muita coisa. Por isso, sobrescrevemo-lo para dizer exatamente o que queremos que o construtor dos nossos Widgets faça. No caso acima, utilizamos a sobrescrita para inserir o `Scaffold` e iniciar a parte gráfica da nossa aplicação.

Dentro do método `build` retornamos uma verdadeira árvore de elementos. Widget por Widget construímos a nossa aplicação. Dentro do `Scaffold` especificamos a nossa `appBar` e o `body` . A `appBar` é a barra superior da nossa tela, ou, para alguns, `toolbar` . No `appBar` podemos personalizar totalmente a nossa barra com texto, ícones, cores, tamanho, funcionalidades e afins.

Fica a gosto do freguês. E no `body`, inserimos todos os Widgets que queremos na nossa tela. No caso, utilizamos primeiramente o Widget `Center` para centralizar todos os elementos filhos. Depois, criamos uma coluna utilizando a propriedade `mainAxisAlignment` para alinhar todos os elementos filhos ao centro. Posteriormente, temos os nossos componentes de texto.

```
floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        child: Icon(Icons.add),
        backgroundColor: Colors.red,
      ),
```

O nosso `Scaffold` também aceita elementos "irmãos" no `body`. Um exemplo de elemento irmão que estamos utilizando é o `floatingActionButton`. Os irmãos são elementos que se posicionam no mesmo nível hierárquico dos outros de mesma importância. Ele está no mesmo nível do Widget `Center`. Com o `floatingActionButton` temos um botão flutuante na tela totalmente personalizável. Podemos alterar o posicionamento, cor, ícone, tamanho e funcionalidade. O nosso botão flutuante contém três especificações: `onPressed` (função que será executada quando pressionarem o botão), `child` (qual será o elemento filho, no nosso caso é um ícone) e `backgroundColor` (cor do botão).

Agora, estamos quase acabando! No rodapé da nossa página, queremos alguns botões que executem uma ação sempre que o usuário tocar neles. Para isso, criaremos a nossa `bottomNavigationBar`.

```
      bottomNavigationBar: BottomAppBar(
        color: Colors.brown,
        child: Container(
          height: 100,
```

```
          child: Padding(
            padding: EdgeInsets.all(20),
            child: Row(
            children: <Widget>[
              IconButton(
                icon: Icon(Icons.add_box, color: Colors.white),
                onPressed: () {
                  alert('Adicionei qualquer coisa');
                }
              ),
              IconButton(
                icon: Icon(Icons.add_a_photo, color: Colors.white
),
                onPressed: () {
                  alert('Adicionei uma foto');
                }
              )
            ],
          ),
          )
        ),
      ),
    );
  }
}
```

O `bottomNavigationBar` é praticamente a mesma coisa que a `appBar`, o que difere é o posicionamento. Como diz o próprio nome do elemento, ele se localiza na parte bottom, ou, no rodapé. Dentro da nossa barra, especificamos que a cor dela será marrom e que o filho dela será um `container`. Dentro do `container` inserimos todos os elementos que precisamos começando com um `Padding` para dar distância das "paredes", assim, centralizando melhor a nossa linha que contém os `IconButtons`. Cada `IconButton` tem a propriedade `onPressed`, que, executa uma função quando leva um tap, ou, "click".

```
void alert(String message) {
  print(message);
}
```

E, por último, mas, não menos importante... O método `alert` . A cada tap em um dos dois `IconButton` da `bottomNavigationBar` ele cria um print no console de desenvolvimento (não aparecerá nada no aplicativo, apenas na IDE/Editor de texto). A seguir está um print com o console logando o `print` que demos ao pressionar os `IconButtons` .

```
Launching lib/main.dart on iPhone in debug mode...
Automatically signing iOS for device deployment using specified development team in Xcode project:
M7B5
Xcode build done.                                           28,3s
flutter: Adicionei qualquer coisa
flutter: Adicionei uma foto
```

Figura 4.9: IconButtons console

Caso queira ver este projeto na íntegra, o código está disponível no endereço:

https://github.com/leomhl/flutterbook_scaffold/

4.5 STATEFUL HOT RELOAD

A cada compilação com tempo médio de um minuto em outras tecnologias, você perde pelo menos cinco minutos no processo todo. Lembra todas as vezes que você mandou executar o compilador para alterar a posição de um botão apenas dois pixels para cima e foi pegar café ou ler as notícias ou responder alguém em alguma rede social? Pois é, o programa gastou um minuto, você, cinco e a sua atenção e foco, quinze, para voltarem a funcionar. Não tente negar, eu sei que é assim, desenvolvo aplicativos desde quando o Android e o iOS eram "tudo mato". O tempo de compilação dos aplicativos com certeza diminuiu absurdamente desde os dispositivos móveis ancestrais, e os

envolvidos nessa melhora estão de parabéns. Para manter a produtividade dos programadores e do desenvolvimento, as atualizações precisam ser realizadas em bem menos de um minuto, mais precisamente em pouquíssimos segundos.

Uma das funcionalidades fascinantes do Flutter é o `hot reload` e sua incrível velocidade durante o desenvolvimento, tirando o máximo de proveito da compilação `JIT` para isso. As alterações geralmente levam menos de um segundo entre salvar que você executou e a exibição na tela do emulador ou smartphone. Vale frisar mais uma vez que o estado da aplicação não muda quando o `hot reload` é ativado assim que você modificar algo no código. Se a sua variável `nome` estiver armazenando "João", essa string se manterá na variável. Não é necessário digitar tudo novamente sempre que alterar algo.

O que podemos concluir quanto à arquitetura do Flutter?

Todo o milagre e mágica na verdade não é milagre nem mágica. Tudo é uma questão de boa estruturação, tipagem bem definida e uma linguagem realmente preparada para a proposta do framework. Widgets estratégicos como o `Scaffold`, `FloatingActionButton`, o mecanismo de `hot reload`, a organização das pastas com menos arquivos e mais centralização das funcionalidades em Widgets colaboraram em demasia para o bom funcionamento do ecossistema como um todo. Em resumo, todo o desempenho e facilidades se dão ao modelo arquitetônico no qual o Flutter foi fundamentado e desenvolvido.

Capítulo 5

A BASE DO FRAMEWORK, DART!

5.1 A CEREJA DO BOLO

Vários linguistas acreditam que a linguagem natural que uma pessoa fala afeta a forma como ela pensa, o mesmo conceito se aplica ao mundo eletrônico. Os programadores que trabalham em diferentes tipos de linguagens de programação geralmente apresentam soluções radicalmente diferentes para os problemas que ocorrem.

O que isso tem a ver com Flutter e Dart? Na verdade, muito. A equipe de engenheiros do Flutter avaliou de início mais de uma dúzia de linguagens e escolheu o Dart porque combinava perfeitamente com a maneira como eles queriam criar as interfaces de usuário e o motor das aplicações.

Dart é AOT (*Ahead Of Time*), compila todo o código para linguagem de máquina antes da execução, trazendo estabilidade e previsibilidade ao comportamento do código. A compilação Ahead of Time proporciona um tempo de inicialização mais rápido principalmente quando a maior parte do código é executada na inicialização, ou seja, quando o aplicativo é aberto. O Dart também

pode ser configurado para trabalhar em modo JIT (*Just In Time*) interpretando para ciclos de desenvolvimento que exijam uma atualização dos componentes na tela de forma excepcionalmente rápida, sem a necessidade de recompilar tudo.

A linguagem facilita a criação de animações e transições suaves que são executadas a 60 fps (frames por segundo). Ela pode fazer alocação de objetos e coleta de lixo (*garbage collector*) sem bloqueios, aliviando a carga da memória e disco de uma forma muito transparente e simples.

Com o Dart, o Flutter não tem a necessidade de uma linguagem de layout declarativo separado, como JSX, XML, HTML, ou construtores de interface visual separados, porque o layout declarativo e programático da linguagem é fácil de compreender e visualizar sem precisar separar tudo em camadas.

E com todo o layout em uma linguagem só e em um único lugar, é fácil para o Flutter fornecer ferramentas avançadas que facilitam a construção e gerência do layout das aplicações. Tudo fica mais centralizado, o que facilita o desenvolvimento e a manutenção.

Quando começamos a estudar a linguagem descobrimos que ela é particularmente fácil de aprender porque possui recursos que são familiares aos usuários de linguagens estáticas e dinâmicas.

Nem todos esses recursos são exclusivos do Dart, mas, a combinação deles atinge um ponto ideal que torna a linguagem excepcionalmente poderosa para implementar o Flutter. Sendo assim, é bastante difícil imaginar o Flutter sendo tão poderoso como é sem estar nos ombros desse gigante chamado Dart.

5.2 ENTENDENDO A ARQUITETURA DART

Historicamente as linguagens de programação são divididas em dois grupos básicos, as estáticas e as dinâmicas. Um grande exemplo de linguagem estática e que eu acredito que todos conheçam é o C. Suas variáveis e constantes são estaticamente tipadas em tempo de compilação. Já as linguagens dinâmicas, como o PHP e o JavaScript, podem ter os tipos dos objetos e variáveis modificados em tempo de execução. Apesar da não flexibilidade das linguagens estáticas, seus códigos são compilados para o Assembly, ou algo mais próximo possível do código de máquina para que, quando forem executados, o próprio hardware entenda as instruções e as cumpra sem a necessidade de intermediários. Com isso, a performance é otimizada ao máximo. As linguagens dinâmicas perdem nesse quesito. Apesar de facilitar em demasia a vida dos desenvolvedores, elas necessitam de um intermediário para entender os seus códigos e passar as instruções para o hardware - entenda "intermediário" como as máquinas virtuais e/ou os interpretadores.

Toda essa teoria que estamos vendo é realmente bastante legal e didática, mas, na prática nada é tão simples. Você deve conhecer o conceito de máquina virtual, ou, popularmente conhecido como VM. Basicamente, uma máquina virtual é um software de alta complexidade que consegue simular as respostas das chamadas de hardware como se fosse o próprio. Com elas, temos uma camada de compatibilidade entre o nosso código e o hardware real, afinal, não trabalharemos com o real e, sim, com o simulado. Como funciona essa bruxaria toda? Vamos lá! Usando o Java como exemplo que dispõe da máquina virtual JVM (Java Virtual Machine) podemos entender melhor esses conceitos. Basicamente,

o desenvolvedor cria o código Java que, quando é executado, realiza uma compilação para uma linguagem intermediária conhecida como bytecode e este, sim, é entendido e executado pela JVM.

Para você entender melhor quais são as grandes vantagens do Dart para o Flutter, você precisa entender melhor sobre compiladores. Para isso, vamos nos debruçar sobre o que falamos anteriormente sobre JIT e AOT. Os compiladores JIT executam as modificações do seu código em tempo de execução, ou seja, é a mesma coisa que trocar os pneus de um ônibus andando, ou as asas de um avião voando. Parece algo impossível, mas os compiladores conseguem realizar tal feito com maestria. Compiladores AOT trocam os pneus e as asas quando o ônibus e o avião estão estacionados, desligados e em solo (isso é ótimo, principalmente para o ônibus! Não queremos ônibus voadores sem rodas por aqui [calma, foi piada]).

Em geral, apenas as linguagens estáticas podem trabalhar com a compilação nativa de máquina, ou AOT justamente por terem tipos fixados e sem variação durante a execução. Então, aqui, fixamos a ideia de que um código mais bem estruturado e tipado no final das contas executa de forma mais leve e rápida.

Mas e as linguagens dinâmicas? É simples! Não são rodadas com o AOT e sim com JIT. Entende agora a razão de o JavaScript, PHP, Java e amigos não executarem tão rápido quanto o C e outras linguagens estáticas? Essa parte a maioria dos professores não contam na faculdade.

Ok, até aqui entendemos que a compilação AOT traz mais previsibilidade e velocidade, mas, isso em tempo de

desenvolvimento ou em produção? Nos dois! E isso é bom e ruim. Quando a compilação AOT é feita durante o desenvolvimento, ela invariavelmente resulta em ciclos de desenvolvimento muito mais lentos (o tempo entre fazer uma mudança no código e poder executá-lo para ver o resultado). Mas esse tipo de compilação resulta em programas que podem ser executados de forma mais previsível e sem pausa para análise e recompilação em tempo de execução como o JIT faz.

Por outro lado, a compilação JIT fornece ciclos de desenvolvimento muito mais rápidos, mas pode resultar em execução mais lenta ou irregular. Em particular, os compiladores JIT possuem tempos de inicialização mais lentos, porque, quando o programa é executado, o compilador precisa fazer uma análise e uma compilação completa de tudo antes que o código possa ser executado. Posteriormente, ele analisa e recompila apenas a parte que foi alterada.

Vários estudos já mostraram que muitas pessoas abandonam aplicativos lentos quando o assunto é a inicialização. Mas acredito que não precisamos de estudos para comprovar isso, aplicativos com inicialização lenta são extremamente irritantes para qualquer um principalmente se estivermos com pressa. Eu nunca fui satisfeito com os híbridos justamente por isso, o tempo de inicialização deles leva em média quase 7 segundos, isso se o aplicativo utilizar o carregamento preguiçoso (*lazy load*) para as telas. Se não, aumente em pelo menos mais cinco ou dez segundos. Chega uma hora em que essa demora se torna inadmissível e isso resulta na desistência do uso e uma posterior desinstalação.

Antes de trabalhar no Dart, os atuais membros da equipe de

engenharia do Google fizeram um trabalho inovador em compiladores avançados e máquinas virtuais, tanto para linguagens dinâmicas (como o mecanismo V8 para JavaScript) quanto para linguagens estáticas. Eles usaram essa experiência para torná-lo excepcionalmente flexível na forma como pode ser compilado e executado. Dart é uma das poucas linguagens (e talvez a única) preparada para ser executada tanto em AOT quanto JIT. O suporte a ambos os tipos fornece vantagens significativas para a linguagem e especialmente ao Flutter.

A compilação JIT é usada durante o desenvolvimento por ser ótima para gerar de forma rápida alterações em tempo de voo (lembra da troca de pneu com o ônibus andando? Então, é isso). Quando um aplicativo está pronto para ser lançado (modo de produção), é compilado em AOT tendo em vista que o código não será modificado enquanto o usuário o está utilizando. Consequentemente, com a ajuda de ferramentas avançadas e compiladores, o Dart pode oferecer o melhor dos dois mundos: ciclos de desenvolvimento extremamente rápidos e tempo de execução e inicialização em produção igualmente rápido.

A flexibilidade da linguagem na compilação e execução não para por aí. Por exemplo, o Dart pode ser transpilado em JavaScript para que possa ser executado pelos navegadores. Isso permite a reutilização de código entre aplicativos móveis e aplicações web. Ele também pode ser usado em um servidor, seja compilado para código nativo ou transformado em JavaScript rodando em cima do nosso querido Node.js.

Por fim, essa verdadeira maravilha do mundo da programação também fornece uma VM autônoma que usa a própria linguagem

Dart como sua linguagem intermediária (atuando como um intérprete). Em resumo, o Dart pode ser eficientemente compilado em AOT ou JIT e interpretado ou transpilado. Não só a compilação e execução do Dart são extraordinariamente flexíveis, como também são especialmente rápidas e é aqui que está a cereja do bolo que tornou a linguagem a eleita para abraçar o Flutter.

5.3 A PONTE QUE ESTAMOS DERRUBANDO USANDO AOT

Pontes geralmente são excelentes. Ligam cidades, estados, países, ruas e casas. É através das pontes que as pessoas conseguem ir e voltar do trabalho, viajar, encontrar quem não viam há anos e muitas outras histórias lindas ou trágicas. A verdade é que a criação de um modelo de "passagem suspensa" por um lugar que não era tão simples de transitar tornou possível a nossa locomoção e otimização de tempo. Poupou-nos de percorrermos grandes distâncias, mas, tem seus preços. O primeiro é o financeiro, para construir algo é necessário investir. O segundo é o trabalho humano para projetar, construir e testar (espero que ainda estejam fazendo pontes com esses três itens e nessa ordem). E o terceiro é o engarrafamento. Reparou que nas redondezas das pontes geralmente existe lentidão? No instante em que este parágrafo é escrito estou em um ônibus preso no engarrafamento na entrada da ponte Rio-Niterói. Se tem algo que faz muito sentido para mim é que as pontes reduziram, sim, os abismos e a distância, mas, são um caminho bem mais lento, complexo, perigoso e concorrido.

Se no mundo físico as coisas são assim, por qual razão no digital seriam diferentes? A terra dos bits e bytes é apenas uma

representação abstrata do mundo em que vivemos, só que em forma digital. Já falamos bastante da capacidade do Dart para transformar por meio do AOT o código em instruções diretas para a máquina. O código compilado em AOT é altamente mais previsível que os JIT justamente por não haver pausas na execução para analisar o contexto e recompilar algo.

Se você se esqueceu da diferença de JIT e AOT, vou dar mais uma explicação muito rápida! Se você quisesse construir um prédio com AOT, caso algo quebrasse ou precisasse ser substituído, você poderia alterar o projeto, demolir e refazer do zero com a alteração (entenda refazer como recompilar o seu código todo). Assim, é garantia total de que o prédio será previsível e que respeitará o esperado no projeto. Se você precisasse trocar uma coluna de sustentação com JIT, não seria necessário demolir e reconstruir, bastaria escorá-lo com toras de madeira, quebrar a coluna antiga, construir a nova e retirar as toras. Funciona? Para o mundo digital, sim, e fica bom? Fica! Pode dar problema? Com certeza! Na hora em que as toras de madeira estão fazendo o papel de uma coluna de cimento não pode haver nenhuma anomalia como terremotos, enchentes, trepidações ou excesso de peso não previsto no projeto do prédio.

Mas Leo, e se um cenário atípico desses acontecer? Lembra de quando eu contei que um aplicativo híbrido meu por algum erro de execução um belo dia liberou permissão que o usuário não tinha para ver as telas? Então... É disso para pior o que pode acontecer com o cenário atípico no momento das toras. Coisa que, se você reconstruir o prédio do zero, não ocorre. Por isso é tão importante AOT para a produção e JIT apenas para desenvolvimento.

Além de todas as vantagens anteriores, voltando ao que estávamos falando, existe algo muito interessante também ao utilizar AOT, que é justamente fugir da "ponte do JavaScript". Nos aplicativos híbridos com JavaScript, é necessário o uso de uma ponte para se comunicar com o código nativo e o grande problema disso é a troca de contexto das informações ao mesmo tempo em que se mantém o estado da aplicação.

Para o correto funcionamento da ponte, os valores das variáveis e parâmetros passados para uma chamada nativa são salvos através de bancos de dados locais (sim, em arquivos de dados ou pequenos bancos que atuam nesse meio de campo). Acho que agora você consegue identificar o local exato em que essa ponte cria o engarrafamento. Basta uma pequena falha nessa ponte e receberemos uma gigante mensagem de erro que geralmente descreve muitas coisas que não dizem nada ao programador e que não ajuda a resolver o problema.

Figura 5.1: Ponte

5.4 CONCORRÊNCIA, DIVISÃO DE TEMPO E RECURSOS COMPARTILHADOS

A maioria das linguagens computacionais que suportam várias threads de execução simultânea (incluindo Java, Kotlin, Objective-C e Swift) usam preempção para alternar a execução entre os processos. Cada thread recebe uma “fatia” de tempo para executar

as instruções que necessitam no processador e, se excedê-lo, a thread é precedida por uma alternância de contexto, dando lugar a outra thread e assim o ciclo de processamento acontece, como um gigante revezamento.

No entanto, se a preempção ocorre quando um recurso que é compartilhado entre processos (como memória) está sendo atualizado, isso causa uma condição de concorrência de recurso. As condições de corrida são um golpe duplo, pois podem causar sérios erros incluindo o travamento de seu aplicativo e a perda dos dados. São particularmente difíceis de localizar e corrigir, pois dependem do tempo consumido pelas threads independentes até mesmo de outros processos que não têm a ver com os do seu aplicativo ou sistema. É muito comum que as condições de corrida parem de se manifestar quando você executa o aplicativo em um depurador e é aí que mora o grande problema.

A maneira típica de consertar uma condição de corrida é proteger o recurso compartilhado usando um bloqueio que impeça a execução de outros segmentos de código, mas os próprios bloqueios podem causar travamentos ou até mesmo problemas mais sérios que exijam o encerramento da execução da aplicação.

O Dart teve uma abordagem diferente para este problema. As threads em Dart são chamadas de forma isolada e não compartilham memória, o que evita a necessidade da maioria dos bloqueios. A linguagem também isola a comunicação transmitindo mensagens através de canais, o que é bastante semelhante ao que os *web workers* em JavaScript fazem. Dart, assim como o JavaScript, é de thread única o que significa que não permite preempção. Em vez disso, as threads geram algumas coisas que

nós, programadores Dart, gostamos muito de usar cotidianamente, que são os `async/await`, `Futures` ou `Streams`. Isso nos dá mais controle sobre a execução como um todo. O encadeamento único nos ajuda a garantir que as funções críticas (incluindo animações e transições) sejam executadas até a sua conclusão, sem preempção, ou seja, sem pausas e longas trocas de contexto através do hardware para que outro processo de outro programa reveze o processador com nosso aplicativo. Isso geralmente é uma grande vantagem que acelera a execução das instruções e reduz consideravelmente os erros por potenciais falhas nas trocas de contexto.

5.5 INTERFACE UNIFICADA

Para mim, a interface unificada veio para ficar. É um grande benefício do Dart o fato de o funcionamento de o Flutter não dividir o código em camadas de layout como JSX, XML ou HTML, nem exigir ferramentas para o desenvolvimento visual separado. O Dart não trabalha com a marcação e estilização separada da parte principal para criar as suas funcionalidades, tudo é escrito com uma só linguagem, Dart! Aqui está um código bem curtinho para você entender melhor isso:

```
Column(children: [
    Text('Olá mundo!', style: TextStyle(fontSize: 80)),
    Icon(Icons.android, color: Colors.green, size: 80),
])
```

E qual é o resultado dele?

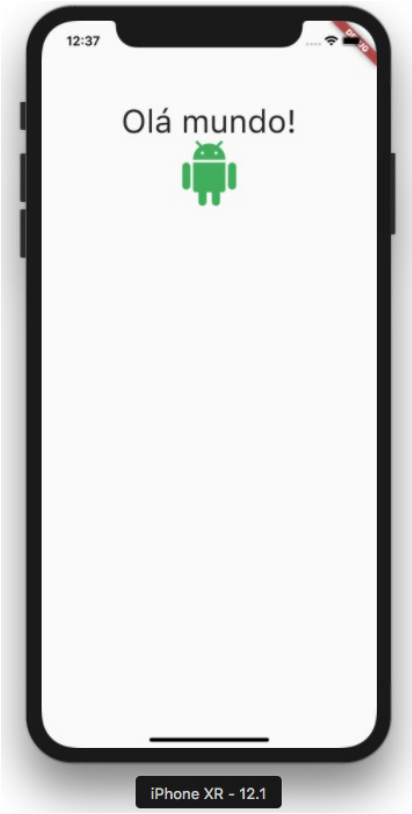

Figura 5.2: Olá mundo

É bastante simples entender o código que criamos em Dart e comparar com o resultado criado via Flutter. Basicamente usamos alguns Widgets aqui, sendo o de centralização, coluna, texto e ícone. Viu como é simples? Não existem as camadas web de estrutura, layout e lógica (HTML, CSS e JavaScript). É tudo criado e especificado em Dart.

Claro que este exemplo com o texto, ícone, padding e toolbar não utilizou apenas os códigos que eu citei. O arquivo `dart.main`

completo que utilizei foi este:

```
import 'package:flutter/material.dart';

void main() {
  runApp(
    MaterialApp(
      home: Scaffold(
        body:
        Center(child:
          Padding(
            padding: EdgeInsets.fromLTRB(0, 100, 0, 0),
            child: Column(children: [
              Text('Olá mundo!', style: TextStyle(fontSize: 80)),
              Icon(Icons.android, color: Colors.green, size: 80),
            ])
          )
        )
      )
    )
  );
}
```

Caso não entenda algo deste código, fique calmo que no decorrer do livro você entenderá melhor tudo isso.

5.6 ALOCAÇÃO DE MEMÓRIA E CUIDADOS COM O LIXO

Outra causa séria de travamentos e lentidões ou até mesmo quebra na execução de um sistema é a coleta de lixo. Na verdade, esse é apenas um caso especial de acesso a um recurso compartilhado (memória), que em muitas linguagens requer o uso de bloqueios. Mas os bloqueios podem impedir que todo o aplicativo seja executado enquanto a memória é limpa pela coleta de objetos e variáveis que estão instanciadas, mas que não são utilizadas. No entanto, o Dart pode executar a coleta de lixo quase

o tempo todo sem criar bloqueios de execução.

Ele usa um esquema avançado de coleta e alocação de lixo que é particularmente rápido para buscar muitos objetos de vida curta (perfeito para interfaces de usuário reativas como Flutter, que reconstroem a árvore de visão imutável para cada quadro). Dart pode alocar um objeto com um único salto de ponteiro (sem necessidade de bloqueio). Mais uma vez, isso resulta em rolagem e animações extremamente suaves, sem perdas e principalmente sem travamentos.

Então, podemos concluir que a arquitetura com que a linguagem Dart foi construída forneceu toda a flexibilidade e robustez necessária para que tecnologias derivadas fossem desenvolvidas de forma realmente confiável e aplicável.

CAPÍTULO 6

WIDGETS, WIDGETS POR TODA A PARTE!

6.1 STATELESS WIDGET

O Flutter contém alguns nomes diferentes para quem vem do mundo Web. Alguns são simples de compreender e outros nem tanto. Vamos ver agora o que é um `stateless Widget` da forma mais simples possível. Na prática, são `Widgets` sem estado. O `stateless Widget` não possibilita alterações dinâmicas, entenda-o como algo completamente estático.

Os `stateless Widget` são amplamente utilizados para a criação de estruturas estáticas nos nossos aplicativos (telas, menus...), tudo o que não envolva *inputs* dos usuários, acessos a APIs e coisas mutáveis. Imagine-os como tijolos em uma construção: são a base mas não fazem nada além disso. Já uma lâmpada, por exemplo, tem a função de ligar e desligar. No caso, uma lâmpada seria um `stateful Widget` , mas esse é um papo para os próximos parágrafos. A seguir, vemos alguns exemplos de Widgets imutáveis.

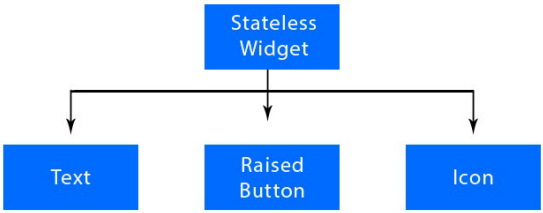

Figura 6.1: Widgets stateless

Em resumo, tudo o que será estático no seu código pode ser considerado `stateless`. E tudo o que depender de alguma interação do usuário para alterar algo não é `stateless`. Esse conceito será importante para os próximos parágrafos.

Mas como criamos um `Widget stateless`? É bem simples, e já fizemos isso em exemplos anteriores. Que tal construirmos juntos algo `stateless`? Todo `Widget` precisa ser declarado como uma classe e estender a classe `StatelessWidget` ou `StatefulWidget`. E todo `Widget` tem um método chamado `build` que retorna os nossos elementos que formam o componente como um todo. Para iniciarmos o processo, vamos criar as declarações básicas para iniciar qualquer projeto, ok? Mãos à obra!

```
import 'package:flutter/material.dart';

void main() {
 runApp(FriendsApp());
}

class FriendsApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     title: 'Meus amigos',
     home: Scaffold(
       appBar: AppBar(
```

```
        title: Text('Meus amigos'),
      ),
      body: Center()
    ),
  );
 }
}
```

Acabamos de criar o código que será o coração da aplicação. Na parte que deveria receber os `Widgets` para a tela exibir algo, inserimos um `Center()`, ou seja, um `Widget` vazio apenas para não dar erro por enquanto. Logo começaremos a criar coisas incríveis utilizando-o.

Um elemento vazio não é o aplicativo dos nossos sonhos, então, que tal colocar um elemento de texto com algum nome? Para isso, precisamos especificar para o `Center` que vamos criar um elemento filho.

```
import 'package:flutter/material.dart';

void main() {
 runApp(FriendsApp());
}

class FriendsApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     title: 'Meus amigos',
     home: Scaffold(
       appBar: AppBar(
         title: Text('Meus amigos'),
       ),
       body: Center(
         child: DecoratedBox(
           decoration: BoxDecoration(color: Colors.lightBlueAccen
t),
           child: Text('Maria'),
         ),
       ),
```

```
      ),
    );
  }
}
```

Dissemos ao `Center` que o `Widget` `DecoratedBox` é o `Widget` filho dele, e definimos uma decoração, `BoxDecoration`, especificando que a cor de fundo será azul. Por último, especificamos que o filho do `DecoratedBox` será um elemento `Text`. Agora, em teoria temos um elemento de texto estilizado e centralizado bem no meio da tela do aplicativo. Vamos executar o código para ver se funcionará corretamente? A imagem a seguir exibe o resultado do código que acabamos de criar.

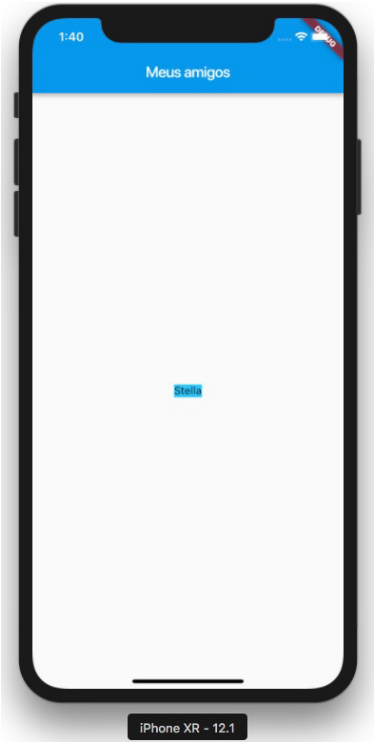

Figura 6.2: Widget Stateless

E se quiséssemos criar vários `Widgets` em vez de um só para a nossa lista de amigos? Certamente ficariam grudados em cima uns dos outros. Precisamos utilizar o `Widget Padding` para nos ajudar a dar uma margem de distância entre os elementos. Também, analisando o código anterior, podemos reparar que o `Widget Center` aceita apenas um `Widget` como filho, então, como podemos inserir vários outros?

Com a estrutura atual é fácil perceber que não podemos inserir mais de um `Widget` filho, então, precisaremos utilizar outro `Widget` chamado `Column` para criar a nossa coluna e alinhar tudo o que precisamos ao centro.

Vamos fazer exatamente isso agora, gerar uma forma de criar "N" elementos filhos em vez de apenas um.

```
import 'package:flutter/material.dart';

void main() {
 runApp(FriendsApp());
}

class FriendsApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     title: 'Meus amigos',
     home: Scaffold(
       appBar: AppBar(
         title: Text('Meus amigos'),
       ),
       body: Center(
         child: Column(
           mainAxisAlignment: MainAxisAlignment.center,
           children: [
             Padding(
                padding: const EdgeInsets.all(10),
                child: DecoratedBox(
```

```
                  decoration: BoxDecoration(color: Colors.lightBl
ueAccent),
                  child:  Text('Maria'),
                ),
              ),
               Padding(
                 padding: const EdgeInsets.all(10),
                 child: DecoratedBox(
                  decoration: BoxDecoration(color: Colors.lightBl
ueAccent),
                  child:  Text('Gabriel'),
                ),
              ),
            ],
          ),
        ),
      ),
    );
 }
}
```

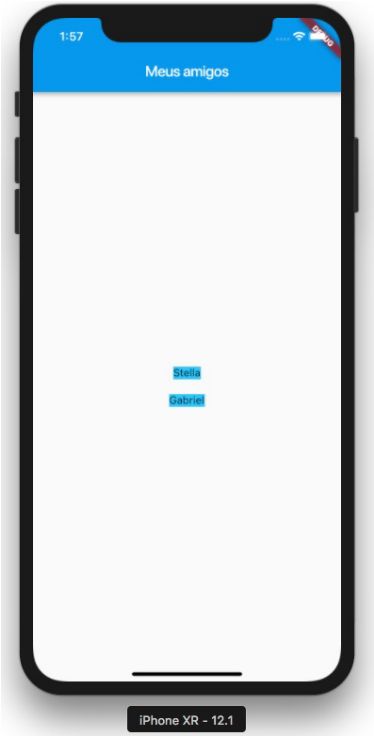

Figura 6.3: Widget Stateless, amigos

Pronto! Temos dois Widgets e estão com um espaçamento entre um e outro atribuídos pelo `Widget Padding`. Mas como somos detalhistas e bastante profissionais, queremos que a fonte do texto fique maior. Para isso, podemos atribuir um estilo aos `Widgets Text` da seguinte maneira:

```
Text('Maria', style: TextStyle(fontSize: 50))
```

Realizando essa alteração em ambos os elementos `Text`, temos a fonte aumentada para 50 pixels e o resultado é o seguinte:

Figura 6.4: Widget Stateless, fonte maior

Bem melhor agora, não é? Quem utiliza óculos, como eu, agradece! E se, quisermos inserir mais amigos? Agora vai ficar divertido! É só replicar várias vezes esses Widgets. Veja o exemplo:

```
import 'package:flutter/material.dart';

void main() {
 runApp(FriendsApp());
}

class FriendsApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
```

```
      title: 'Meus amigos',
      home: Scaffold(
        appBar: AppBar(
          title: Text('Meus amigos'),
        ),
        body: Center(
          child: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: [
              Padding(
                 padding: const EdgeInsets.all(10),
                 child: DecoratedBox(
                   decoration: BoxDecoration(color: Colors.lightBl
ueAccent),
                   child:  Text('Maria', style: TextStyle(fontSize
: 50)),
                 ),
              ),
               Padding(
                 padding: const EdgeInsets.all(10),
                 child: DecoratedBox(
                   decoration: BoxDecoration(color: Colors.lightBl
ueAccent),
                   child:  Text('Gabriel', style: TextStyle(fontSi
ze: 50)),
                 ),
              ),
               Padding(
                 padding: const EdgeInsets.all(10),
                 child: DecoratedBox(
                   decoration: BoxDecoration(color: Colors.lightBl
ueAccent),
                   child:  Text('Vanessa', style: TextStyle(fontSi
ze: 50)),
                 ),
              ),
               Padding(
                 padding: const EdgeInsets.all(10),
                 child: DecoratedBox(
                   decoration: BoxDecoration(color: Colors.lightBl
ueAccent),
                   child:  Text('Camila', style: TextStyle(fontSiz
e: 50)),
                 ),
              ),
```

```
          ],
        ),
      ),
    ),
  );
 }
}
```

Ficou bem legal, mas não parece muito inteligente repetir tantas vezes o bloco do `Padding` para alterar apenas o texto do elemento `Text`. E se pudéssemos criar uma estrutura genérica de um `Widget` e chamá-lo quando quisermos passando o nome do amigo em vez de todas as vezes repetir a estrutura toda? Mas, como fazer isso? Simples! Dê uma olhada no código:

```
import 'package:flutter/material.dart';

void main() {
 runApp(FriendsApp());
}

class FriendsApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     title: 'Meus amigos',
     home: Scaffold(
       appBar: AppBar(
         title: Text('Meus amigos'),
       ),
       body: Center(
         child: Column(
           mainAxisAlignment: MainAxisAlignment.center,
           children: [
              MyFriend('Maria'),
              MyFriend('Gabriel'),
              MyFriend('Anderson'),
              MyFriend('Camila'),
           ],
         ),
       ),
     ),
```

```
   );
 }
}

class MyFriend extends StatelessWidget {
    final String text;
    const MyFriend(this.text);

  @override
  Widget build(BuildContext context) {
    return Padding(
            padding: const EdgeInsets.all(10),
            child: DecoratedBox(
              decoration: BoxDecoration(color: Colors.lightBlueAc
cent),
              child:  Text(text, style: TextStyle(fontSize: 50)),
            ),
          );
  }
}
```

Criamos um `Widget` totalmente novo, chamado `MyFriend`. Estamos utilizando os nomes em inglês apenas por convenção - você pode chamar do que quiser, até mesmo em português. Utilizamos o `Widget MyFriend` várias vezes no `body` da aplicação passando apenas o nome do amigo. Acabamos de economizar uma grande quantidade de código em um exemplo simples, mas imagine isso em um aplicativo realmente grande como uma loja virtual, por exemplo. A personalização de `Widgets` é fantástica e merece o nosso carinho.

Vamos desmembrar o `Widget MyFriend` para entender melhor:

```
class MyFriend extends StatelessWidget {
    final String text;
    const MyFriend(this.text);

  @override
  Widget build(BuildContext context) {
```

```
    return Padding(
            padding: const EdgeInsets.all(10),
            child: DecoratedBox(
              decoration: BoxDecoration(color: Colors.lightBlueAc
cent),
              child:  Text(text, style: TextStyle(fontSize: 50)),
            ),
          );
  }
}
```

Como foi visto anteriormente, um `Widget` é criado como uma classe que estende, neste caso, o `StatelessWidget`, já que não há nada dinâmico dentro dele. Como o `Widget` não terá estado, ou seja, será estático, após a sua criação utilizaremos uma constante final (que é imutável após receber um valor) em vez de uma variável para tornarmos o `Widget Text` reaproveitável. Para pegar o valor que vem da chamada do `Widget`, utilizamos o próprio construtor da classe, como podemos ver:

```
final String text;
const MyFriend(this.text);
```

Feito isso, a constante está carregada com o valor que queremos (`Maria`, por exemplo). Posteriormente, sobrescrevemos o método `build` que vem da classe `StatelessWidget` e personalizamos do jeitinho que quisermos. Vale lembrar que o método `build` obrigatoriamente precisa que um `Widget` seja retornado. Nosso aplicativo criado 100% com `Widgets Stateless` ficou assim:

Figura 6.5: Widgets Stateless finalizados

Caso queira ver este projeto na íntegra, o código está disponível no seguinte endereço:

https://github.com/Leomhl/flutterbook_stateless/

6.2 STATEFUL WIDGET

Os `Widgets stateful` são praticamente o oposto dos `stateless` . Eles contêm estado e são mutáveis. É por meio deles que construiremos boa parte das aplicações e componentes. São elementos-chaves para o desenvolvimento móvel da forma

interativa que conhecemos. A criação deles pode parecer um pouco mais complexa que a dos `stateless`, mas estudando com calma passo a passo como faremos, você vai entender tudo perfeitamente. Ambos os tipos de `Widgets` são bastante simples e tranquilos de implementar, é só você sempre se perguntar se é hora de utilizar `stateless` ou `stateful` e seguir adiante.

Na criação de um `Widget stateless`, precisamos basicamente de uma classe estendendo a classe `StatelessWidget` e, dentro dela, implementar o método `build`, retornando à nossa árvore de `Widgets` que formam o novo `Widget` personalizado. Já para os `stateful Widgets` precisamos criar mais um passo: temos que pensar também no estado.

O estado de um elemento basicamente controla o que é mutável nele. Sempre que uma variável, parâmetro ou similares mudam, automaticamente podemos comunicar ao Flutter que houve essa mudança e ele executará uma reconstrução do `Widget` alterado, atualizando na tela a exibição do elemento com o valor novo.

Vamos para a prática? Criaremos um contador bastante básico. Ele exibirá um nome e o número de toques que ocorreram na tela. É básico, porém, é tudo o que precisamos para entender o conceito de `stateful`. Mãos à obra!

A criação do `stateful Widget` tem uma diferença crucial: o código estrutural do `Widget` não fica no `Widget`, e sim, no estado! Afinal, é ele que comandará os valores mutáveis dos nossos elementos. Segue o exemplo da criação do `Widget ItemCounter`:

```
class ItemCounter extends StatefulWidget {
```

```
  final String name;
  ItemCounter(this.name);
}
```

Basicamente recebemos um nome que será estático nele e pronto, é isso. Mas, e a parte em que contaremos os toques na tela? Vamos criá-la no estado, afinal é algo dinâmico, concorda? Para criar o estado que queremos, precisamos fazer exatamente da seguinte forma:

```
class _ItemCounterState extends State<ItemCounter> {
  int count = 0;

  @override
  Widget build(BuildContext context) {
    return Text('${Widget.name}: $count', style: TextStyle(fontSi
ze: 50));
  }
}
```

Criamos uma classe privada, observe pelo `_` antes do nome `_ItemCounterState`. A classe `_ItemCounterState` estende a classe `State` e estamos passando como tipo o nosso `Widget stateful ItemCounter` para a estrutura do estado ser criada exatamente nos moldes que precisamos para o elemento. Mas se você prestou bastante atenção, não tem nada ligando o `Widget` ao estado. Então, o que precisamos fazer agora é chamar o estado dentro do `Widget` para que a mágica ocorra! Para isso, a classe `StatefulWidget` implementa um método chamado `createState` que vamos sobrescrever criando um método com o mesmo nome dentro do `Widget` para iniciar a conexão entre ele e o estado.

Até este momento, o aplicativo está mais ou menos assim:

Figura 6.6: Contador que não conta nada

Como o nome da imagem diz, nosso aplicativo contador é maravilhoso mas não conta absolutamente nada! Precisamos capturar os eventos de *tap* (toque) na tela para começar a contagem na variável `count`. Para isso, utilizaremos o `Widget GestureDetector` que o Flutter já traz em seu SDK. O estado ficará assim:

```
class _ItemCounterState extends State<ItemCounter> {
  int count = 0;

  @override
  Widget build(BuildContext context) {
    return GestureDetector(
```

```
      onTap:() {
        count++;
      },
      child: Text('${Widget.name}: $count', style: TextStyle(font
Size: 50))
    );
  }
}
```

Agora sim, tem que funcionar! Ops, ainda não funciona. Você pode tocar a tela diversas vezes que nada acontecerá, mas por qual motivo? Simples, recebemos o disparo do evento tap, incrementamos mais um na variável `count` mas não informamos ao estado do `Widget` que houve mudanças e que ele precisa atualizar os elementos. Para isso, precisamos utilizar o método `setState` da seguinte forma:

```
class _ItemCounterState extends State<ItemCounter> {
  int count = 0;

  @override
  Widget build(BuildContext context) {
    return GestureDetector(
      onTap:() {
        setState((){
          count++;
        });
      },
      child: Text('${Widget.name}: $count', style: TextStyle(font
Size: 50))
    );
  }
}
```

Como gostamos de testar bastante as coisas para ver se funcionam, demos dez toques na tela e o nosso aplicativo ficou assim:

Figura 6.7: Contador que conta bem pra caramba

Parabéns, você acabou de criar o seu primeiro `Widget stateful` ! Existe uma forma mais inteligente, arquiteturalmente falando, para avisar a troca de estado dos `Widgets` que não utiliza o método `setState` . Como o intuito deste livro é ensinar o caminho para iniciar no Flutter, não explicarei essa técnica tendo em vista que é bem mais complexa e avançada. Mas caso tenha interesse, pesquise por `Widget Streams` e `BLOC pattern` .

Para ver o projeto na íntegra, o código está disponível em: https://github.com/Leomhl/flutterbook_contador/

6.3 MAIS UM POUCO DE TEORIA SOBRE WIDGETS PARA EXPLODIR A SUA MENTE

E se eu lhe contasse que todo `Widget stateful` no fim do processo se torna um `Widget stateless`, você se assustaria? Então... Oficialmente é isso que acontece. O que vemos impresso na tela dos celulares ou tablets é totalmente `stateless`, mesmo tendo sua criação `stateful`. Mas, ainda assim, é necessário criar `Widgets stateful` para qualquer coisa que exija mutabilidade. Lembra-se do método `build` do `state`? Pois é, o que ele retorna como produto final é um elemento `stateless` e isso faz muito sentido.

Quer entender melhor isso? Pense em uma folha de papel impressa, não precisamos alterar o que tem escrito nela depois da impressão, mas, quando estávamos editando o conteúdo que seria impresso era muito importante a mutabilidade. Para editar o texto utilizamos o computador (máquina extremamente complexa e completa) e quando está tudo pronto transformamos em algo bem mais simples para a leitura, daí se dá a impressão da folha. Você não precisa de um computador para apenas olhar um texto, mas não consegue editá-lo tendo apenas um papel, entende? Assim são os `Widgets`. Segue um exemplo visual:

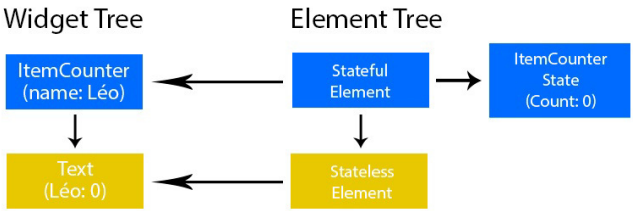

Figura 6.8: Diagrama stateful 1

Como você pode ver, a árvore de `Widgets` não precisa ter o estado de cada Widget, apenas a representação visual dele. Quem cuida do estado é o próprio `Widget` e ele não precisa, nem deve, repassar isso para ninguém. Já imaginou se a `AppBar` precisasse se preocupar com o estado de um campo de texto lá na quinta tela de um aplicativo? Seria um verdadeiro inferno em forma de código. Ainda bem que criaram a arquitetura das aplicações para nos poupar dessa burocracia loucamente desnecessária.

Cada `Widget` cuida de seu estado e fornece para a árvore principal de `Widgets` um elemento `stateless`. Mas para onde vai o valor das variáveis que alteramos? Continuam quietinhas dentro do `Widget stateful` na parte de estado bem longe da árvore `stateless`, a única coisa que muda no processo é o que será enviado para ser exibido na tela. O que era 0 vira 1 quando damos um toque na tela, seguindo a lógica do exemplo dado no capítulo anterior. Comparando o diagrama a seguir com Diagrama stateful 1 você verá a diferença que ocorre por baixo dos panos.

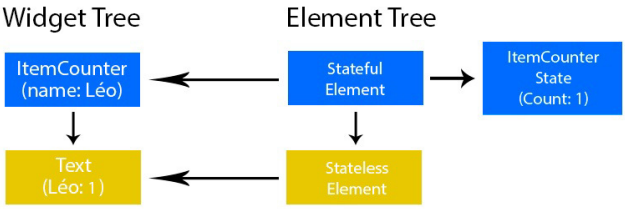

Figura 6.9: Diagrama stateful 2

Cotidianamente falando, se o `stateful` se torna `stateless` pouco importa, afinal, é algo totalmente transparente ao desenvolvedor, mas é bastante interessante entender o conceito e como de fato funciona todo o processo.

Capítulo 7

DEPENDÊNCIAS

As dependências podem ser entendidas como tudo o que o aplicativo precisa de adicional para cumprir com suas funcionalidades ou layout. Elas podem ser pacotes de código criados na linguagem Dart que adicionam funcionalidades extras ao projeto, como animações personalizadas ou a capacidade de vibrar o dispositivo, por exemplo. Também podemos considerar como dependências arquivos de imagem, vídeos e o que mais o desenvolvedor precisar inserir no projeto. Em resumo, são coisas extras que podem agregar valor ao projeto.

Bons exemplos de dependências amplamente utilizadas são as que possibilitam a leitura de códigos de barras, compressão e descompressão de arquivos, acesso a localização via GPS, acesso ao leitor biométrico (caso o dispositivo tenha este sensor) e acesso ao flash. O Google sozinho não consegue criar todas as dependências de que precisamos, afinal, a demanda é bastante específica para cada desenvolvedor e o número de projetos cresce a cada dia.

Pensando nisso, eles disponibilizaram meios para que a comunidade de desenvolvedores crie e publique gratuitamente dependências no repositório aberto da linguagem Dart. Para saber mais você pode acessar a página de desenvolvedores Flutter que se encontra na seguinte URL:

https://flutter.dev/docs/development/packages-and-plugins/developing-packages/.

Mas em qual site podemos procurar o que já existe de dependências disponíveis e como podemos adicionar ao nosso projeto? Basta acessar https://pub.dartlang.org/flutter/. É importante lembrar de filtrar as extensões dizendo que você está buscando apenas as que são para o Flutter, ok? Caso contrário, a busca retornará todas as extensões disponíveis que batem com o termo buscado, não necessariamente somente as para o Flutter. Ao acessar essa URL, você verá uma tela igual ou parecida com a seguinte imagem:

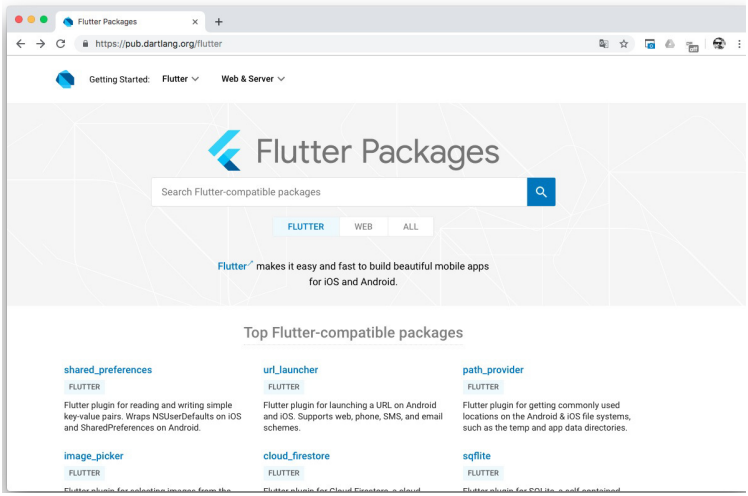

Figura 7.1: Flutter packages

Que tal criarmos um aplicativo bastante básico que realiza a leitura de QR Codes, apenas para entendermos melhor a utilização de uma dependência?

Caso não saiba, o QR Code ou, *Quick Response Code* é um código de barras bidirecional. Diferente dos códigos de barras antigos que proporcionavam apenas o armazenamento de números, com o QR Code tornou-se possível transformar dados alfanuméricos em um código facilmente interpretável por leitores e câmeras. O uso de QR Codes é livre de qualquer licença, e é definido e publicado como um padrão reconhecido pela ISO. Sendo assim, por ser padronizado, é bastante simples encontrar extensões que leiam praticamente qualquer tipo de QR Code.

Para começar, crie um projeto do zero com o nome "qr_code". Depois, vá ao endereço https://pub.dartlang.org/flutter/ e pesquise pelo termo "qrcode_reader". Provavelmente você encontrará uma tela similar a esta:

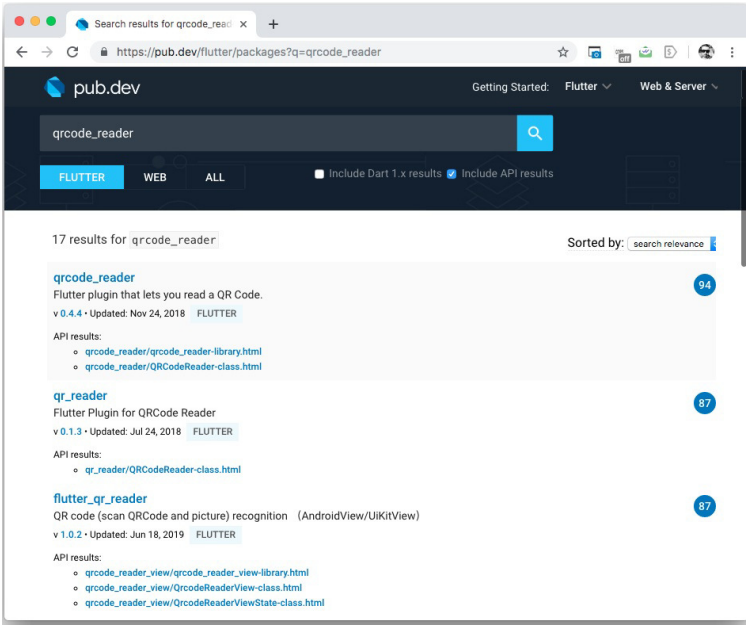

Figura 7.2: Dart QR Code package

Utilizaremos a extensão `qrcode_reader` na versão 0.4.4 que é a mais atualizada durante o momento em que este livro é escrito. Caso a versão seja mais nova no momento em que você estiver lendo, utilize a mais atual verificando na documentação na página da extensão se os comandos são os mesmos e adapte de acordo com as suas necessidades. Abrindo a página da extensão `qrcode_reader`, encontramos a seguinte tela:

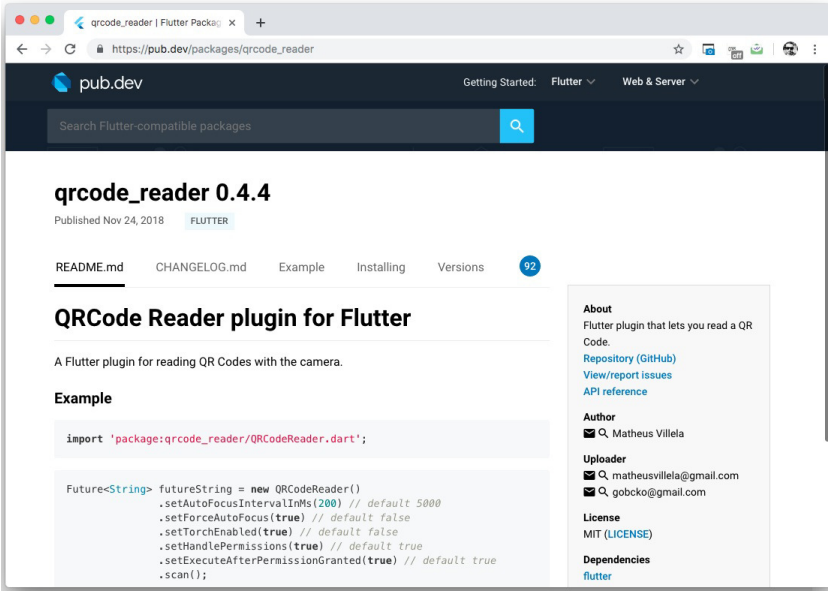

Figura 7.3: Página do qrcode_reader

Por padrão, a tela é aberta na aba `README.md` que ensina como utilizar a extensão, porém, antes de usar precisamos adicioná-la ao projeto. Para isso, vamos abrir a aba `Installing`. Nela encontramos tudo o que é necessário para instalar e incorporar a extensão ao projeto.

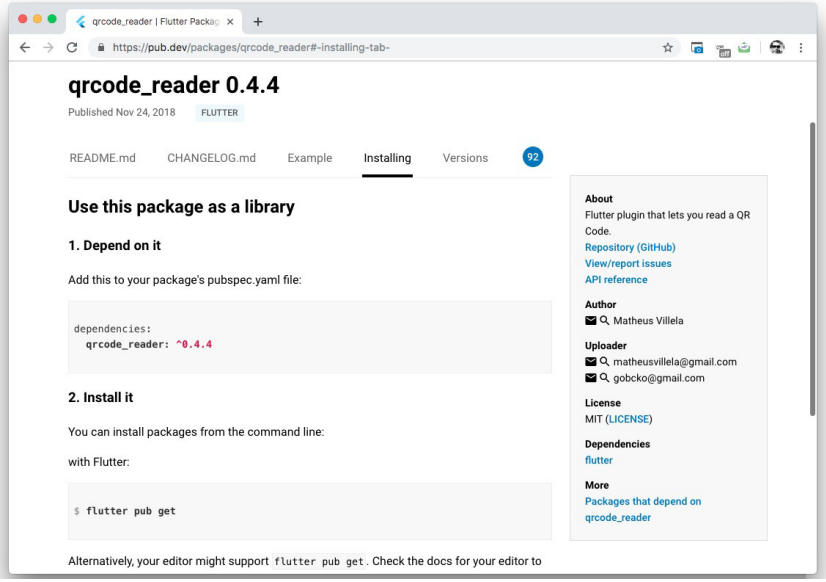

Figura 7.4: Aba de instalação

No item "1. Depend on it", copie apenas a descrição da dependência:

```
qrcode_reader: ^0.4.4
```

Cole no arquivo `pubspec.yaml` na parte de *dependencies* (abaixo do `cupertino_icons`). Cuidado com a indentação, ela é muito importante. O arquivo `pubspec` é totalmente mapeado pela indentação realizada com os *tabs*. Caso indente errado, dará erro. Quando salvar a alteração no arquivo a sua IDE ou editor de texto vai atualizar e baixar as novas dependências (ou pelo menos deveria). Caso não baixe de forma automática, utilize no terminal/shell/bash o seguinte comando: `flutter packages get`. É necessário que o terminal esteja no diretório da pasta do projeto `qr_code`.

O seu arquivo `pubspec.yaml` deverá ficar similar a:

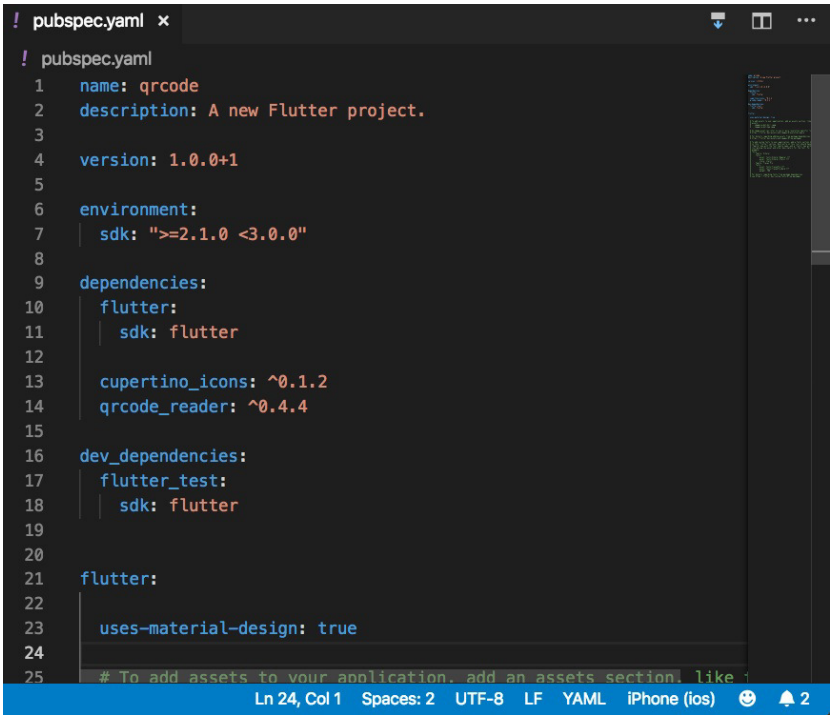

Figura 7.5: Dependência qrcode_reader no pubspec

Feito isso, podemos começar a utilizar o leitor de QR Codes no arquivo `main` . Limpe todo o conteúdo do arquivo `main.dart` que veio por padrão quando o projeto foi criado. A primeira coisa que devemos inserir nele são os *imports*.

```
import 'package:flutter/material.dart';
import 'package:qrcode_reader/qrcode_reader.dart';
```

O *import* referente ao `material.dart` é padrão, todo projeto precisa ter. Já o segundo, você encontra na aba *Installing* na seção "3. import it".

Após importar as dependências necessárias, vamos iniciar o funcionamento do projeto com a função `main`. Abaixo dos *imports*, insira a seguinte linha:

```
void main() => runApp(MyApp());
```

Pronto, podemos começar a escrever o `StatelessWidget` que estruturará a página. Abaixo da linha da chamada da função `main`, crie a primeira parte do `Widget` com o seguinte código:

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Leitor de QR Code',
      theme: ThemeData(

        primarySwatch: Colors.blue,
      ),
      home: MyHomePage(title: 'Leitor de QR Code'),
    );
  }
}
```

Feito isso, temos a estrutura básica da página. Porém, ainda falta dar vida para a nossa criação: falta a criação do `StatefulWidget`. Abaixo da declaração da classe `MyApp`, crie a classe `MyHomePage`, que será responsável pelo controle do estado da nossa aplicação (conceito que abordamos no capítulo referente aos `Widgets`).

```
class MyHomePage extends StatefulWidget {
  MyHomePage({Key key, this.title}) : super(key: key);

  final String title;

  @override
  _MyHomePageState createState() => _MyHomePageState();
}
```

E por último, mas não menos importante, o estado que é o responsável pela funcionalidade de abertura do leitor de QR Code e exibição do resultado. Como o método `_MyHomePageState` ficará bem grande e pouco didático para explicar por inteiro, vamos dividir em três partes, sendo elas: `_captureQR` , `_showDialog` e `build` . Então, abaixo da classe `MyHomePage` insira as seguintes linhas de comando:

```
class _MyHomePageState extends State<MyHomePage> {

  void _captureQR() {

  }

  void _showDialog(String qrtext) {

  }

  @override
  Widget build(BuildContext context) {

  }
}
```

O método `_captureQR()` será responsável por chamar o leitor de QR Codes, que abrirá a câmera do dispositivo e ficará analisando a imagem e aguardando detectar um código QR nela. Assim que o código for encontrado, ele vai chamar o método `_showDialog` passando como parâmetro a `String` que a decodificação do código gerou. Veja como o método `_captureQR` é criado:

```
  void _captureQR() {

    Future<String> futureString = QRCodeReader().scan();
    futureString.then((qrText) {
      _showDialog(qrText);
    });
```

```
}
```

O método `_showDialog`, por sua vez, apresenta uma janela de diálogo com o título "Texto do QR Code". O conteúdo será a `String` que está no QR e, nas ações (*actions*), criaremos um botão para fechar a janela de diálogo. A seguir, o código de como o método `_showDialog` é criado:

```
void _showDialog(String qrtext) {
  showDialog(
    context: context,
    builder: (BuildContext context) {
      return AlertDialog(
        title: new Text("Texto do QR Code"),
        content: new Text(qrtext),
        actions: <Widget>[
          new FlatButton(
            child: new Text("Fechar"),
            onPressed: () {
              Navigator.of(context).pop();
            },
          ),
        ],
      );
    },
  );
}
```

Finalmente, temos o método `build`, que constrói o estado do `Widget`. É nele que a exibição dos `Widgets` do corpo da tela serão criados e disponibilizados para o usuário operar o aplicativo. Veja como implementar o método `build`:

```
@override
Widget build(BuildContext context) {

  return Scaffold(
    appBar: AppBar(
      title: Text(widget.title),
    ),
    body: Center(
```

```
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          RaisedButton(
            child: Text('Ler QR Code'),
            onPressed: () {
              _captureQR();
            },
          )
        ],
      ),
    ),
  );
}
```

Ao executar o código, o aplicativo gerado ficará assim:

Figura 7.6: App leitor de QR Codes

Caso queira ver este projeto na íntegra, o código está disponível no seguinte endereço:

https://github.com/Leomhl/flutterbook_qr_code/

Que tal brincarmos um pouco? Através do site *QR Code Generator* (https://br.qr-code-generator.com/) podemos criar um código QR para testar se tudo funcionou corretamente. Utilizaremos a `String` "Desenvolva aplicações móveis no Dart Side!".

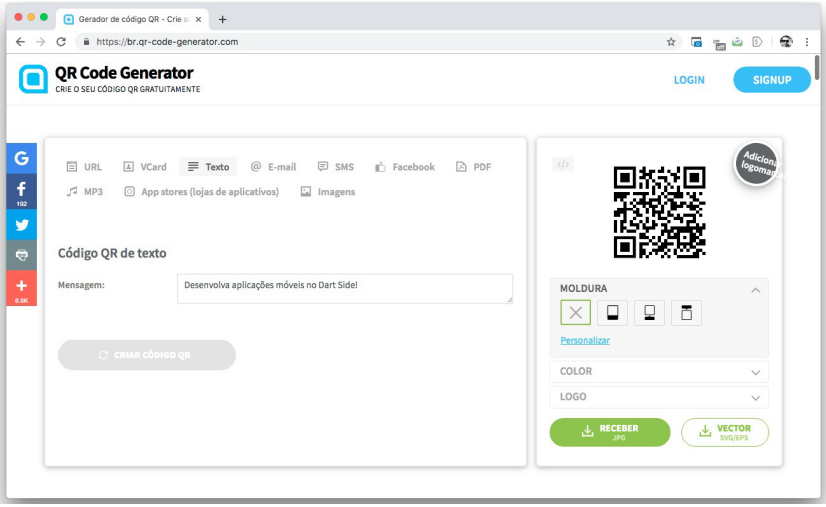

Figura 7.7: QR Code personalizado

Ao ler o QR Code com o nosso aplicativo, o resultado que obtemos é o seguinte:

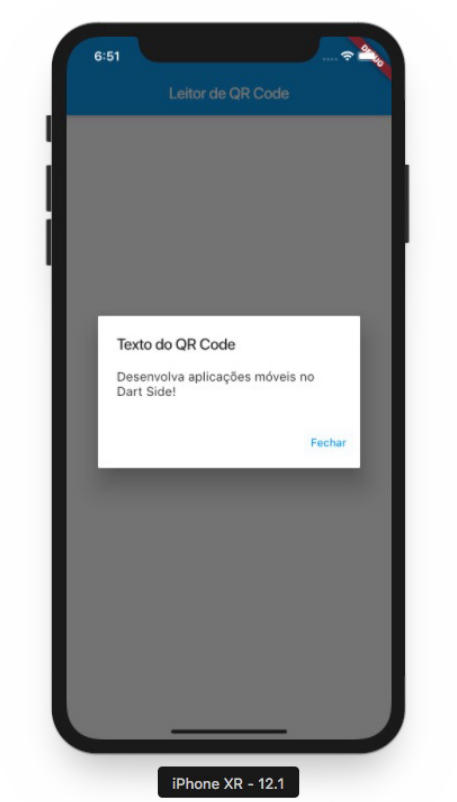

Figura 7.8: QR Code decodificado

Capítulo 8

PROTOTIPAÇÃO

8.1 QUEM SABE PARA ONDE QUER IR CHEGA MAIS RÁPIDO

Você já pegou um ônibus simplesmente para ver aonde ele chegaria? Você se deslocou por vários quilômetros, sem rumo? No fim, a viagem até foi bem-sucedida e você chegou a algum destino, mas muito tempo foi investido e no final o resultado foi algo próximo de "Ah, legal!". Por outro lado, alguém que já sabia qual ônibus pegar para ir direto ao seu destino com certeza foi muito mais assertivo e rápido.

Assim também é o desenvolvimento de aplicativos. Um aplicativo que recebe como feedback "Ah, legal!" tem muito o que melhorar. É preciso pensar antes aonde se quer chegar e planejar seus passos futuros. Em programação, isso equivale à etapa de desenhar, sem julgamentos, afinal, é a versão mais básica e inicial possível do que virá a ser o seu aplicativo. Desenhar o protótipo do que você quer é o primeiro passo para ganhar tempo e entregar algo realmente bonito e útil que não frustre o processo de desenvolvimento.

Nós, desenvolvedores, em boa parte dos casos não temos grandes habilidades em design e usabilidade. Sabemos fazer

funcionar, esse sim é o nosso grande talento. O problema é que desenvolvimento de software é um tema muito amplo, abstrato e complexo, que muitos pesquisadores ainda tentam definir. Ainda é algo bastante novo se comparado com outras profissões que existem há séculos.

O meu conselho é: não use "fazejamento", ou seja, ir programando enquanto planejando o próximo passo sem uma visão ampla do que sairá como produto final. Pense primeiro, execute depois. Não o inverso! E isso não se aplica só a softwares. Quando você sabe o seu destino, traçar a melhor rota se torna algo simples, e no mundo da programação isso não é diferente. Compartilhando um pouco da minha experiência, já perdi dois meses de trabalho em uma empresa tentando montar uma interface agradável, mas, infelizmente, nada que eu fazia parecia agradar os clientes. Comparei aplicativos concorrentes, pesquisei bastante sobre design minimalista e me esforcei muito, mas nada disso adiantou. Então, após os dois meses mais longos da minha vida, sugeri a contratação de um designer. Em uma semana foi entregue um arquivo Photoshop com a prototipação das telas desenhadas de uma forma que eu jamais pensaria. Eu e mais uma amiga desenvolvedora levamos pouco menos que uma semana para implementar as sugestões do novo *layout* e, como mágica, algo que não saiu em dois meses ficou maravilhoso em duas semanas. Ganhamos tapinha nas costas e vastos elogios, além de uma caixinha de cápsulas para a nossa cafeteira como mimo.

Quero dizer que você só vai conseguir criar aplicativos lindos, práticos e maravilhosos com um profissional para desenhá-los para você? Não! Mas, que facilita muito, com certeza facilita. Após este episódio, comecei a investir mais em formas de entender como

prototipar o que eu quero fazer. Se desde o começo eu prototipasse em vez de perder semanas codando uma interface que não seria aprovada, com certeza não teríamos perdido dois meses. E lembre-se, no mercado de trabalho tempo vale dinheiro e a paz interior do desenvolvedor. Lição aprendida! Pesquise, inspire-se e divirta-se! Torne esse processo quase terapêutico. Programar por si só é algo complexo e muitas vezes monótono. A parte de prototipação precisa ser divertida e leve. Existem vários sites como o *dribbble* (https://dribbble.com/) que dão sugestões altamente inspiracionais e modernas de *layouts* para aplicativos.

8.2 CUIDADO COM O FRANKENSTEIN

Lembra do fazejamento? Pois é, é aqui que o Frankenstein aparece e faz o aplicativo com um potencial maravilhoso se tornar uma verdadeira aberração. Apenas se inspirar em *layouts* bonitos e montar "de cabeça" o que você quer, achando que vai ganhar tempo com isso, é pura enganação. É aqui que a perda de tempo acontece. É necessário prototipar desenhando o que quer para ter uma referência visual. Às vezes, a ideia de um menu maravilhoso com uma barra inferior animada quando colocados juntos ficam feios, mesmo que separados em outros aplicativos sejam bonitos. Coisas bonitas isoladas nem sempre se mantêm assim juntas. Tudo na mente fica maravilhoso, mas quando colocamos no papel realmente temos um parâmetro de como ficará de forma real, e pode ser bem diferente da expectativa.

A grande problemática que desperta todos os sentimentos de repulsa e preguiça nos programadores é o tenebroso Photoshop ou programas similares. Pensando nisso, a Adobe (não recebo para fazer a propaganda) criou um software muito interessante e

simplíssimo de utilizar que possibilita prototipar sites, aplicativos e uma série de outras coisas de maneira bastante simples. Também conseguimos criar algumas animações e transições de tela com ele para auxiliar na validação se a interface será satisfatória o suficiente para ser programada. Tal programa milagroso se chama Adobe XD.

Encontrou no *dribbble* várias telas legais? Escolha as de que mais gostou, junte os "retalhos", desenhe em uma folha de papel (feito é melhor do que perfeito) e veja o que achou. Se aparentou ser bom, vá para o XD e prototipe para tirar do papel e ter algo realmente próximo do desejado. A prototipação não sairá perfeita, afinal, não somos especialistas nisso, mas é um excelente começo. Quem sabe para onde quer ir não perde tempo desbravando rotas longas e ruins! Caso não utilize um computador compatível com o pacote Adobe, como os computadores de sistema Linux você pode optar pelo *Figma*, ele tem a finalidade bastante próxima do XD e é totalmente online (web). De qualquer navegador moderno é possível acessar e prototipar coisas através dele. Também é uma alternativa bastante interessante.

Um pouquinho mais sobre o XD

De acordo com o site do Adobe XD, ele é descrito da seguinte maneira:

"Com o Adobe XD, sua equipe cria experiências, protótipos interativos, compartilha-os com revisores, faz iterações e recebe atualizações em tempo real, tudo isso na mesma solução. E, como o XD é integrado ao Photoshop e ao Illustrator, todo o processo de design é concluído com muito mais rapidez. O Adobe XD

aumentou em 10 vezes a produtividade em algumas situações rotineiras de fluxos de trabalho que foram analisadas."

E sabe o que é mais legal? No momento em que este livro é escrito, o XD é totalmente gratuito! Recomendo fortemente que assista algumas aulas sobre prototipação e XD no YouTube ou plataforma que preferir. Existem muitas aulas e tutoriais gratuitos disponíveis focados em ajudar iniciantes. E, acredite, prototipar é algo muito simples de fazer. É só praticar um pouco que dentro de poucas semanas você estará dominando a técnica.

O bom imperador tem sempre um bom plano

Dart é uma linguagem bastante estruturada que permite ao desenvolvedor manter organização e concisão no código. Claro que, como em qualquer linguagem, ainda é possível fazer as coisas de forma desorganizada e problemática, mas o importante é buscar sempre aperfeiçoar o que está criando para tornar-se simples e objetivo. E o Dart nos ajuda nisso.

Assim como nos filmes de guerras, o bom imperador sempre calcula riscos, caminhos, métodos, estratégias, tempo e principalmente os custos para cada decisão tomada. Toda ação é meticulosamente pensada antes de ser executada para garantir o maior sucesso possível com o mínimo de esforço e tempo desperdiçados. Em resumo, prototipar é isso. Você é o imperador dos projetos que cria, então aja como um estrategista e planeje para não desperdiçar o seu tempo e dinheiro.

Grande parte dos softwares criados estouram prazos, orçamentos e contratos justamente por não serem previamente prototipados. Pior do que isso é depois de pronto desagradar ao

cliente que esperava uma interface com poucos botões e você entregou um "painel de avião". Portanto, tenha um bom plano, meça todos os desafios, mostre ao cliente o que pretende fazer e parta para a batalha! Seguindo esses passos as chances de sucesso aumentam e muito!

Capítulo 9

AVANÇANDO COM O FLUTTER

E agora, quais são os próximos passos?

Para os próximos passos precisamos avançar nos conceitos de programação e comunicação com agentes externos. Caso você já venha do mundo da programação e tenha experiência com o desenvolvimento de outros sistemas ou aplicativos, termos como HTTP, JSON, e API não lhe serão estranhos. Caso você não conheça algum desses, aprofundaremos melhor o que cada um é e para o que serve. Este capítulo é parte essencial para o entendimento dos próximos passos para criar uma aplicação mais avançada e que consulte dados de serviços web.

9.1 HTTP

Voltando um passo atrás, para entender o papel do HTTP é necessário entender a razão de ele existir e por que é tão importante para absolutamente tudo no que diz respeito ao tráfego de informações entre nossos aplicativos e dados que vamos buscar de algum lugar. No auge dos anos 70 e 80 não existia um protocolo ou padrão para que todas as máquinas em rede se comunicassem.

Basicamente, cada instituição ou empresa que trabalhava com pesquisas envolvendo computação desenvolvia protocolos internos de comunicação para seus computadores trocarem as informações. Isso posteriormente inviabilizaria a ligação entre diversas universidades e centros de pesquisas para a troca de informações e trabalhos científicos em geral. Perante tal problema, um pesquisador chamado Tim Berners-Lee (mais conhecido como o pai da internet) propôs um padrão de comunicação para a transferência de hipertexto entre computadores de forma comum a todos. Pela primeira vez haveria a ideia de um protocolo de comunicação universal que possibilitou a criação do que conhecemos hoje como internet.

HTTP, ou, *Hypertext Transfer Protocol* é um dos protocolos que regem a internet e a forma como as trocas de informações online são realizadas. Mensagens, imagens, publicações, áudios, vídeos... Tudo isso trafega através de requisições cliente-servidor com HTTP. Ele funciona como um protocolo de requisição-resposta, ou seja, sempre que você pedir algo para algum servidor, ele responderá, ou pelo menos deveria. O oposto também se aplica. Um navegador web, por exemplo, como o *firefox* pode ser o cliente, e uma aplicação em um computador que hospeda um site pode ser o servidor. Um aplicativo Flutter requisitando dados de um serviço que forneça a hora atual pode ser considerado um cliente enquanto o computador que armazena esse sistema que fornece a hora é a máquina servidora. Pode parecer complexo de início, mas é bastante simples e abstrato. Na prática precisamos nos preocupar bem pouco com o protocolo HTTP já que ele faz a maior parte do trabalho pesado da comunicação de forma automática.

O protocolo HTTP implementa alguns "verbos" bastante importante que utilizaremos para consumir um serviço de consulta financeira posteriormente com o Flutter. Os principais e mais utilizados verbos HTTP são `GET` e `POST`. Comumente, `GET` é amplamente utilizado para a busca de informações, isto é, quando desejamos obter algum dado enviamos uma requisição para o lugar que pode nos fornecer tais dados utilizando o verbo `GET`. Já o `POST` é utilizado para o envio de informações por sua segurança em trafegar os dados dentro do corpo da requisição e não no cabeçalho. Ele torna muito mais seguro trafegar dados sensíveis como dados de autenticação, por exemplo.

Todas as linguagens modernas, ou praticamente todas, implementam o conceito de HTTP para acessar a web. Os frameworks criados para estender e facilitar o uso dessas linguagens não são diferentes. No Dart, temos todo um aparato para a utilização de comunicação via HTTP pronto para servir nossas aplicações Flutter de forma altamente simples e segura. Existe tratamento assíncrono para garantir que o tempo de espera entre a requisição e a resposta do servidor não trave a aplicação ou deixe o usuário sem saber o que está acontecendo. É muito importante levar em consideração o tempo de resposta das requisições aos servidores externos já que não considerar tal fato pode encadear uma série de problemas e comportamentos incorretos na execução do código. Nos próximos capítulos veremos na prática a utilização de requisições HTTP.

9.2 API

O termo API diverge bastante em sua descrição. Alguns autores consideram que API é uma sigla para *Application*

Programming Interface (Interface de programação de aplicativos) e outros para *Access point integration* (Ponto de acesso para integração). Na real, uma API é uma espécie de "caminho comum" entre duas aplicações que normalmente é utilizado para realizar a troca de informações (envio e recebimento). É através das APIs que sistemas criados em linguagens, arquiteturas e modelagens completamente diferentes conseguem se comunicar sem nenhum tipo de problema por meio de um padrão de troca de informações comum que ambas entendam.

Normalmente, as APIs comunicam-se utilizando padrões como: SOAP, REST ou GraphQl (claro, não se limita somente a esses, porém são os mais populares). Existem diversos padrões de comunicação e formatos para a troca de dados como JSON ou XML, por exemplo. Na aplicação que criaremos, o consumo de dados será feito de uma API que trabalha no padrão REST trafegando dados em JSON (abordaremos nas próximas seções) para obter informações referentes ao mercado financeiro.

Quando uma pessoa acessa uma página de um restaurante, por exemplo, é possível visualizar dentro do próprio site o mapa do Google Maps para saber a localização do estabelecimento e verificar qual o melhor caminho para chegar até lá. Esse procedimento é realizado por meio de uma API, que os desenvolvedores do site do restaurante utilizam do código do Google Maps para inseri-lo em um determinado local de sua página.

Em resumo, através de uma API, não importa com qual linguagem ela tenha sido feita ou como funciona internamente conseguimos obter dados e enviar dados para ambientes externos.

Apenas o que importa para o bom funcionamento da integração entre sistemas ou para a consulta de informações é o padrão com que os dados trafegam. Se ambas as aplicações trabalharem no mesmo padrão de tráfego de dados, tudo certo. Conseguiremos realizar o nosso trabalho da forma esperada.

9.3 JSON

A notação JSON significa *JavaScript Object Notation*, ou, Notação de Objetos JavaScript. Apesar de atualmente JSON não estar mais atrelado somente à linguagem JavaScript, ainda temos forte referência a ela pela forma com que constrói e exporta seus objetos. Mas por qual razão o JSON é tão difundido e utilizado no mercado? A resposta para esta pergunta é bastante simples até, a notação de objetos JavaScript trouxe um conceito leve para representar objetos que mais tarde se tornou padrão referência para o armazenamento e troca de informações. Basicamente, o JSON reuniu pares de "chave e valor", simplificando absurdamente a representação das informações se comparado aos padrões anteriores como o XML, por exemplo. Com a redução da verbosidade que foi realizada na eliminação das antigas tags de marcação, essa notação tornou-se ideal para trafegar dados em um mundo em que precisamos lidar com a telefonia móvel e ambientes com conexões precárias de rede que necessitam baixar uma gama de informações o mais rápido possível.

Mas por qual razão exatamente não precisamos programar em JavaScript para utilizar JSON? E mais ainda, como utilizamos algo do JavaScript no Dart? Ela é uma notação em formato texto e completamente independente de linguagem. Foi sim criada dentro do JavaScript, mas é apenas uma forma de organizar e representar

dados. Não envolve bibliotecas ou extensões externas. Partindo disso e notando o grande potencial do tráfego rápido e organizado de informações utilizando JSON, as demais linguagens de mercado implementaram métodos para codificar e decodificar objetos JSON, transformando-os em listas/dicionários/arrays dependendo da linguagem e como ela realiza o tratamento dos dados. A seguir, um exemplo de como utilizamos o JSON para a comunicação cliente-servidor.

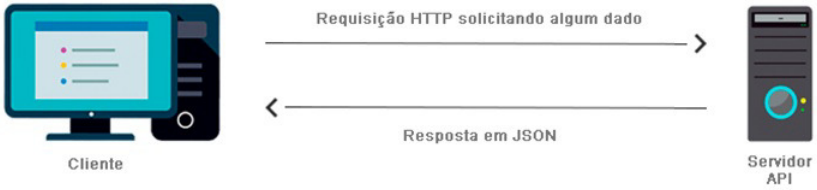

Figura 9.1: Comunicação JSON com web service

A requisição realizada na imagem utiliza o verbo HTTP `GET`, sendo assim, não enviamos dados diretamente para o servidor, caso sejam necessários apenas trafegamos parâmetros ou informações não críticas. Entenda "informação crítica" como senhas, e-mail de usuário e afins. O ideal para trafegar em requisições `GET` são tokens de autenticação e parâmetros específicos como: quantos registros você espera que a busca retorne, identificador de um usuário, por exemplo. O correto para o envio de dados sensíveis é o verbo `POST`; via `GET`, a ideia é apenas obter algumas informações e no máximo passar parâmetros no cabeçalho HTTP ou URL para refinar o resultado da requisição. Ao receber o pedido, o servidor prontamente responde com um objeto JSON contendo tudo o que foi pedido. Adiante podemos

ver como é um objeto em notação JSON.

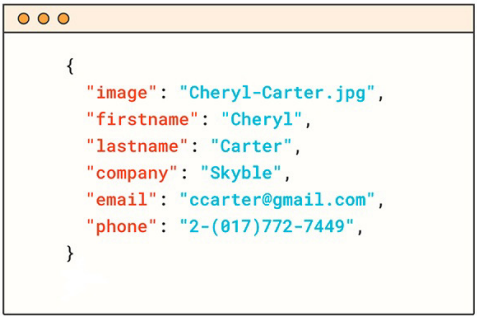

Figura 9.2: Exemplo de objeto JSON

Basicamente, ele é iniciado e encerrado com chaves. Cada dado tem seu par chave e valor separados por : (dois pontos). Cada par é separado por , (vírgula). É possível que um parâmetro tenha como valor outro objeto JSON, assim, conseguimos construir desde passagens simples de dados até matrizes multidimensionais de altíssima densidade.

9.4 COSTURANDO OS RETALHOS

Você já deve ter visto a famosa colcha de retalhos da vovó. É aquele cobertor normalmente flanelado, muito macio e engraçado por ser um todo formado por várias partes. Assim é uma aplicação que utiliza dados externos, uma grande construção formada de várias partes interconectadas. Criaremos um aplicativo unindo os conceitos citados anteriormente, este aplicativo será focado em consultar informações em tempo real da cotação do Dólar, Real e Euro e converter os respectivos valores para a moeda vigente de acordo com a interação do usuário. Para isso, criaremos no nosso

aplicativo Flutter toda a estrutura gráfica e lógica para cumprir esta tarefa. Mas, como funciona isso de exibir um dado que ainda será buscado? Graficamente, criaremos campos "genéricos" que esperam receber os valores vindos de uma API, e tais valores serão entregues para nosso aplicativo em formato JSON. Cabe a nós decodificá-los em mapa do Dart para podermos extrair dentre os diversos dados, os valores de Real, Dólar e Euro e inserir de forma correta nos campos, calculando a devida alteração realizada pelo usuário.

Basicamente, teremos uma estrutura genérica com campos de texto que, quando o usuário digitar algum valor numérico, por meio do protocolo HTTP vai requisitar os valores vigentes da moeda a um servidor que dispõe de tais dados atualizados e exibir ao usuário. Então, para recapitular... O primeiro passo é criar a estrutura gráfica esperando receber valores. A seguir temos um exemplo genérico de como isso funciona.

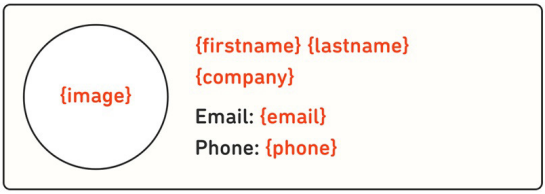

Figura 9.3: Estrutura genérica

Feita a estrutura que espera dados de forma genérica, precisamos nos preocupar em buscar os dados. Para tal, passaremos uma requisição HTTP utilizando o verbo `GET` para buscar os dados sobre cotação de dinheiro do dia na API do `hgbrasil` . Feita a requisição, a API retornará um objeto JSON

com tais informações. Decodificaremos os dados, acessaremos os valores que queremos dentro da gama de valores que a API trará e, por último, faremos o preenchimento dos campos que criamos com tais dados. A seguir, vemos um exemplo ilustrativo do preenchimento dos campos com os valores retornados de uma API de mentirinha.

Figura 9.4: Dados populados

Assimilados os conceitos de HTTP, API e JSON podemos no próximo capítulo dar início à criação do nosso aplicativo conversor de moedas.

Capítulo 10

CRIANDO UM APLICATIVO COMPLEXO

O que entendemos por aplicativo complexo?

Neste capítulo, entenderemos como aplicativo complexo aquele que permite interação com o usuário, comunicação com APIs externas, processamento dos dados vindos de fontes externas junto aos dados inseridos pelo usuário, tratamento de possíveis erros externos e internos ao código da aplicação e, por último, mas não menos importante, que tenha uma interface gráfica amigável aos usuários.

É claro que temos incontáveis maneiras de tornar mais complexo um aplicativo, mas nos ateremos a estes tópicos para tornar o exemplo mais fácil de ser entendido. Utilizaremos todos os conceitos introduzidos no último capítulo. A imagem a seguir demonstra qual será o produto final do aplicativo que criaremos.

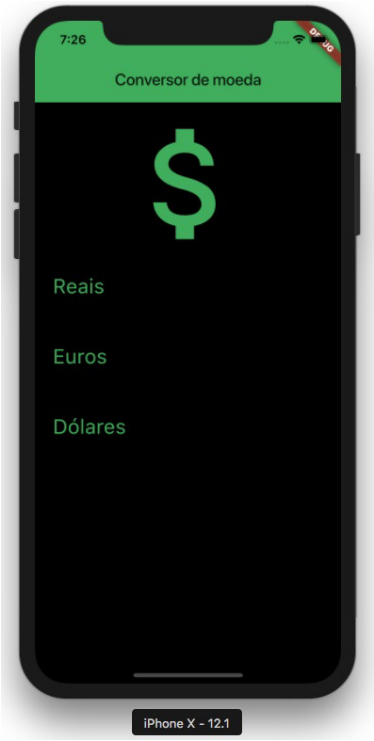

Figura 10.1: Aplicativo conversor de moeda

O usuário poderá informar um valor em real, dólar ou euro e automaticamente o aplicativo vai buscar via uma API os dados do câmbio das três moedas e atualizar as duas outras de maneira a haver equivalência dos valores. A imagem a seguir demonstra como ficará algum dos campos quando o usuário exercer um tap sobre ele. Vale lembrar de que no mundo dos dispositivos móveis não existem cliques, e sim taps!

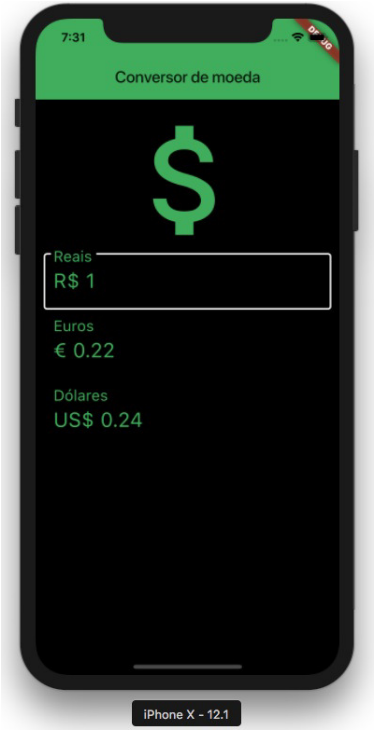

Figura 10.2: Exibição da conversão de moeda

Menos papo, mais código!

Para iniciar o desenvolvimento do aplicativo, precisamos importar algumas bibliotecas fundamentais para o pleno funcionamento do código que escreveremos. Os imports serão realizados seguindo a ordem em que nosso código será criado, então, primeiro teremos a importação do `material.dart` , que é uma biblioteca fundamental para a criação da interface gráfica e que já abordamos amplamente nos capítulos anteriores. Posteriormente, devemos importar a biblioteca que nos serve com o `http` para a realização de requisições a serviços de terceiros, e,

por último, a biblioteca `dart convert` para convertermos os valores vindos da API para mapas (notações JSON) manipuláveis pelo Dart.

```
import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;
import 'dart:convert';
```

Atenção: não se esqueça de importar o http no arquivo `pubspec.yam` como fizemos nos outros exemplos!

Vale prestar atenção no "as" utilizado no `http` . Utilizamos esse comando para criar um apelido para a biblioteca http, assim, conseguiremos utilizar os métodos disponibilizados por ela através desse objeto `http` ao longo do arquivo `main` .

O próximo passo importantíssimo é criar o método `main` , afinal, é ele que vai dar o início na execução do código do aplicativo. Nele, será definido o Widget `Home` , o tema que utilizaremos e as cores que prevalecerão.

```
void main() async {
  runApp(MaterialApp(
    home: Home(),
    theme: ThemeData(
      hintColor: Colors.green,
      primaryColor: Colors.white
    ),
  ));
}
```

Visando respeitar a hierarquia de execução das funções do nosso aplicativo, é importante que a função que vai buscar os dados da API que fornece a cotação das moedas vigentes esteja declarada acima das demais que a invocarão. Para isso, vamos criar antes da implementação do `Widget` o método `getData` , que futuramente vai buscar os dados necessários para a realização das

requisições.

```
Future<Map> getData() async {
  String request = "https://api.hgbrasil.com/finance?format=json&
key=SUA_KEY_AQUI";

  http.Response response = await http.get(request);
  return json.decode(response.body);
}
```

Para este passo, é importante que você crie uma conta na Hg Brasil e gere uma chave de uso da API financeira. É um procedimento totalmente gratuito, e sem essa chave a consulta não será realizada corretamente. O método `getData` retorna um objeto Future através do operador async, ou seja, quando o método for invocado ele realizará a requisição `http` através do método `get` para a API da Hg Brasil e trará os dados como um corpo de requisição.

Para transformar os dados em notação JSON, conhecida em Dart como Mapa, utilizaremos do método `json.decode()` vindo diretamente da biblioteca Dart convert. O próximo passo é criar o `StatefulWidget` que carregará a nossa aplicação.

```
class Home extends StatefulWidget {
  @override
  _HomeState createState() => _HomeState();
}
```

Criado o `StatefulWidget` , é necessário criar o estado da aplicação que guardará as informações que vão tornar nosso aplicativo funcional. Para tal, vamos fatiar em algumas partes o bloco de código do estado para que seja fácil compreender as etapas da programação empregada.

Como vamos criar três campos de texto para exibir o valor das

moedas (Real, Dólar e Euro) e todos eles podem ser alterados pelo usuário, precisaremos criar controladores para eles. É através dos controladores que conseguimos manipular o valor dos campos e detectar alterações realizadas pelo usuário. Vamos iniciar o bloco do estado com a declaração e criação dos controllers dos campos. Também é importante criar uma variável para o cálculo do dólar e outra do euro. Utilizaremos o Real como base, então, não precisaremos de uma variável dedicada para ele.

```
class _HomeState extends State<Home> {

  final realController = TextEditingController();
  final dolarController = TextEditingController();
  final euroController = TextEditingController();

  double dolar;
  double euro;
```

Feito isso, precisaremos de quatro métodos importantíssimos. O primeiro vai limpar todos os três campos (Real, Dólar e Euro) e o chamaremos de `_clearAll`. Ele será utilizado pelos outros três métodos que criaremos, sendo eles: `_realChanged`, `_dolarChanged` e `_euroChanged`. Sempre que um dos campos for alterado, o seu respectivo método `changed` será chamado, e este utilizará do método `_clearAll` para limpar todos os campos antes de inserir os valores atualizados. É de suma importância que esses métodos estejam dentro do estado para que a nossa atual lógica funcione. Segue o código destes métodos.

```
  void _clearAll(){
    realController.text = "";
    dolarController.text = "";
    euroController.text = "";
  }

  void _realChanged(String text){
    if(text.isEmpty) {
```

```
      _clearAll();
      return;
    }
    double real = double.parse(text);
    dolarController.text = (real/dolar).toStringAsFixed(2);
    euroController.text = (real/euro).toStringAsFixed(2);
  }

  void _dolarChanged(String text){
    if(text.isEmpty) {
      _clearAll();
      return;
    }
    double dolar = double.parse(text);
    realController.text = (dolar * this.dolar).toStringAsFixed(2)
;
    euroController.text = (dolar * this.dolar / euro).toStringAsF
ixed(2);
  }

  void _euroChanged(String text){
    if(text.isEmpty) {
      _clearAll();
      return;
    }
    double euro = double.parse(text);
    realController.text = (euro * this.euro).toStringAsFixed(2);
    dolarController.text = (euro * this.euro / dolar).toStringAsF
ixed(2);
  }
```

Cada um desses métodos converte o valor recebido de String para double para realizar o cálculo de quanto a moeda vale perante as demais, e, atribui este valor ao atributo `text` dos `controllers`, assim, o valor em reais com duas casas decimais é enviado para os campos de texto que o usuário verá.

O atributo `toStringAsFixed(2)` serve para garantir que as casas decimais além das duas primeiras serão descartadas já que estamos lidando com valores monetários. Criados os métodos chaves para a manipulação dos campos, é necessário unir todos

eles através do método `build` do `_HomeState` . Para tal, o método `build` deve retornar um `Widget` . No nosso caso retornaremos um `Scaffold` . A primeira parte do método deve ficar assim:

```
@override
 Widget build(BuildContext context) {
   return Scaffold(
     backgroundColor: Colors.black,
     appBar: AppBar(
       title: Text("Conversor de moeda"),
       backgroundColor: Colors.green,
       centerTitle: true,
     ),
```

A seguir, precisamos criar o `body` do `Scaffold` . Para isso, utilizaremos um `Widget` conhecido como `FutureBuilder` . O `FutureBuilder` constrói os elementos perante dados que virão de algo assíncrono, ou seja, ele montará os campos na tela para o usuário ver assim que os dados sobre as moedas cheguem da API. No parâmetro `future` passaremos o método `getData()` para que ele retorne os dados de que precisamos para construir os campos.

```
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      backgroundColor: Colors.black,
      appBar: AppBar(
        title: Text("Conversor de moeda"),
        backgroundColor: Colors.green,
        centerTitle: true,
      ),
      body: FutureBuilder<Map>(
          future: getData(),
```

No `builder` , criaremos um `switch` que será o principal encarregado de testar os estados da conexão. Ele será responsável

por alertar ao usuário de que ele precisa aguardar ou de que tudo já está pronto, e exibe os campos criados.

```
body: FutureBuilder<Map>(
    future: getData(),
    builder: (context, snapshot) {
      switch(snapshot.connectionState){
        case ConnectionState.none:
        case ConnectionState.waiting:
          return Center(
              child: Text("Aguarde...",
                style: TextStyle(
                    color: Colors.green,
                    fontSize: 30.0),
              textAlign: TextAlign.center,)
          );
```

Nos primeiros testes do `switch` ele verifica se a conexão com a api está em "none", ou seja, não aconteceu corretamente, ou se está aguardando o retorno da API (waiting). Para ambos os casos é exibida uma mensagem de "Aguarde..." na tela para dar tempo de a requisição ser realizada.

Quando o status mudar, o `default` do `switch` será acionado. Nele precisamos testar se a resposta da API foi correta e trouxe os dados que esperamos ou se houve algum erro. Caso tenha havido algum erro vamos exibir a mensagem "Ops, houve uma falha ao buscar os dados". Se não, os dados terão vindo corretamente e então vamos atribuir às variáveis `dolar` e `euro` seus respectivos valores tendo em base o Real. Segue o código do default.

```
      default:
        if(snapshot.hasError){
          return Center(
              child: Text("Ops, houve uma falha ao buscar os dado
s",
                style: TextStyle(
```

```
                    color: Colors.green,
                    fontSize: 25.0),
                textAlign: TextAlign.center,)
          );
        } else {
          dolar = snapshot.data["results"]["currencies"]["USD"]["
buy"];
          euro = snapshot.data["results"]["currencies"]["EUR"]["b
uy"];
```

Para encerrar o default do `switch`, retornaremos um Widget `SingleChildScrollView`, que abrigará todos os elementos que serão exibidos na tela para que o usuário consiga ver todos os elementos. Nele, configuraremos o padding dos elementos, alinhamento, ícones, posicionamentos, dividers (para espaçar um campo do outro) e inseriremos as chamadas do método `buildTextField`, que criará os campos de texto. Vamos criar este método para não repetir três vezes a mesma estrutura de código dos campos, para alterar apenas o valor que vai dentro do campo, assim, economizamos código e facilitamos a manutenção posteriormente.

```
        return SingleChildScrollView(
              padding: EdgeInsets.all(10.0),
              child: Column(
                crossAxisAlignment: CrossAxisAlignment.stretch,
                children: <Widget>[
                  Icon(Icons.attach_money, size: 180.0, color: Co
lors.green),
                  buildTextField("Reais", "R\$ ", realController,
 _realChanged),
                  Divider(),
                  buildTextField("Euros", "€ ", euroController, _
euroChanged),
                  Divider(),
                  buildTextField("Dólares", "US\$ ", dolarControl
ler, _dolarChanged),
                ],
              ),
            );
```

```
          }
        }
      })
    );
  }
}
```

Por último, o método que vai criar cada campo de texto. Este método deve ficar após o `_HomeState` . Segue o código do método:

```
Widget buildTextField(String label, String prefix, TextEditingCon
troller c, Function f){
  return TextField(
    controller: c,
    decoration: InputDecoration(
        labelText: label,
        labelStyle: TextStyle(color: Colors.green),
        border: OutlineInputBorder(),
        prefixText: prefix
    ),
    style: TextStyle(
        color: Colors.green, fontSize: 25.0
    ),
    onChanged: f,
    keyboardType: TextInputType.numberWithOptions(decimal: true),
  );
}
```

Ao executar o código construído, o resultado deverá ser o da imagem a seguir. É claro que os valores monetários não serão iguais aos da foto, já que o valor das moedas flutua diariamente, mas a estrutura do aplicativo deverá ficar igual.

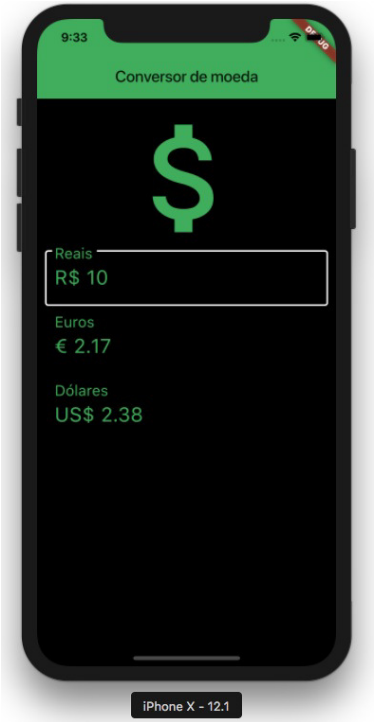

Figura 10.3: Aplicativo conversor de moeda versão final

Para ver o projeto na íntegra, o código está disponível no seguinte endereço:

https://github.com/Leomhl/flutterbook_conversordemoeda

Capítulo 11

BANCO DE DADOS

11.1 UM POUCO SOBRE DADOS

Há quem diga que os dados são o novo petróleo e não é difícil de entender os motivos desta fala. As famosas "Gigantes do vale" dentre as quais podemos citar como o Google (a mãe do Flutter), por exemplo, e empresas concorrentes estão gerando todos os anos bilhões de dólares utilizando em grande parte apenas dados e explorando ao máximo as possibilidades em marketing e Big data que eles oferecem. Hoje, existem muitas empresas que faturam um montante de dinheiro bem maior do que petroleiras de países com um tamanho territorial gigante, como o Brasil.

Para tal movimentação de dados, são necessários servidores dos mais diversos portes para comportar a carga que os bancos de dados armazenam e gerenciam. Bancos de dados são parte fundamental destas operações e da maioria dos sistemas que utilizamos em nosso cotidiano, afinal, são eles que cuidam deste bem tão precioso chamado dado. Um dado não necessariamente diz muitas coisas, mas, aplicando o processamento correto, formatando-o e corrigindo incoerências que possam vir da fonte geradora, obtém-se a informação. Normalmente o que exibimos nas telas de sistemas são informações, em resumo, dados tratados para se tornarem mais palatáveis ao usuário da aplicação.

Para facilitar um pouco a explicação deste conceito, vamos pular para algo mais prático. A data de nascimento de um indivíduo (dia, mês e ano) é considerada um dado; uma informação seria calcularmos a idade dele e exibirmos. Não devemos armazenar a idade atual, afinal, no ano seguinte ela se tornará obsoleta e não saberemos ao certo quando realizar a atualização dela. É importante ter este conceito em mente por mais simples que possa ser, afinal, se salvarmos algo no formato incorreto na nossa futura base de dados, o aplicativo não se comportará como gostaríamos.

Assim como sistemas de grande porte alocados em servidores precisam de bancos de dados para realizar o armazenamento do que virá a ser informação um dia, os aplicativos móveis também precisam. Bancos de dados para aplicativos em grande parte do tempo guardam dados de sessão para garantir que o usuário não precise autenticar-se (login e senha) todas as vezes que abre o aplicativo, dados de navegação, tokens de segurança, dados cadastrados pelo usuário e que se desejam manter localmente (como faremos no exemplo a seguir), criação de cache para a navegação no aplicativo com a ausência de conexão com a internet como as redes sociais fazem, e afins.

São quase infinitas as possibilidades do que podemos armazenar em bancos de dados locais. Claro que dispositivos móveis não dispõem da robustez de hardware necessária para hospedar um banco gigante e altamente performático, e para isso temos os servidores. Nos dispositivos móveis os bancos de dados são "cópias em miniatura" do que os bancos mais robustos, assim, conseguimos garantir a eficiência e segurança de um banco de dados convencional respeitando as limitações do ambiente que o

está hospedando.

Visando segurança, coerência nos dados e velocidade na gravação e recuperação, a comunidade portou a API de acesso de um banco de dados amplamente conhecido para a linguagem Dart, o SQLite. Este banco trabalha utilizando a linguagem SQL para realizar consultas e armazenamento dos dados. Basicamente, o SQLite é uma biblioteca que foi implementada em C visando ser pequeno, demasiadamente rápido, de alta confiabilidade e completo, independente de sistemas gerenciadores de bancos de dados ou burocracias envolvendo o ambiente em que ele trabalha. A parte da alta confiabilidade é bastante importante ressaltar já que bancos de dados para dispositivos móveis no início da "Era Smartphone" eram verdadeiros buracos negros que corrompiam com uma facilidade inacreditável ceifando os dados que estavam neles.

Segundo o site do SQLite https://www.sqlite.org/, ele é o mecanismo de banco de dados mais utilizado no mundo. O SQLite é um banco de dados relacional assim como o MySQL, PostgreSQL e similares. Sendo assim, podemos criar tabelas, consultas com a linguagem SQL e esperar o mesmo grau de confiabilidade e simplicidade dos bancos já conhecidos no cotidiano da maioria dos desenvolvedores.

A extensão criada especialmente para trabalharmos em Dart com o SQLite chama-se **sqflite** e está disponível para download no repositório de pacotes do Dart. Com isso, o Flutter a aceita como dependência e conseguimos trabalhar com este incrível banco de dados em nosso aplicativo.

Iniciaremos criando um aplicativo chamado agenda. Para este

projeto, teremos duas telas: a principal, que listará os dados dos contatos salvos e a tela que utilizaremos para cadastrar ou editar os dados de um contato específico. A seguir você pode ver imagens de como ficará o aplicativo agenda:

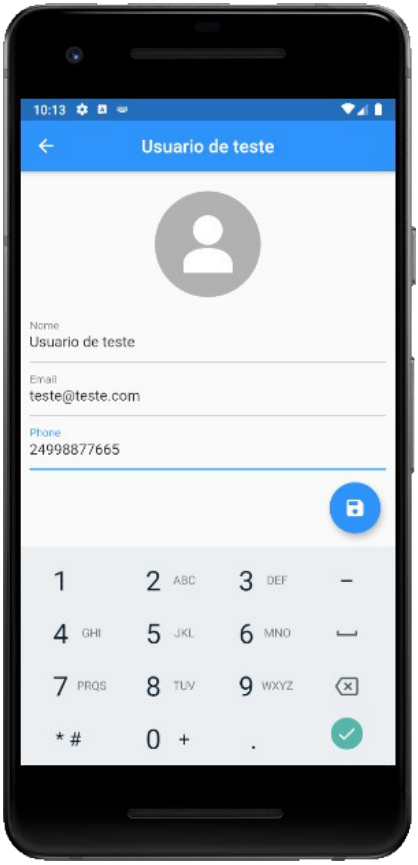

Figura 11.1: Cadastrar contato

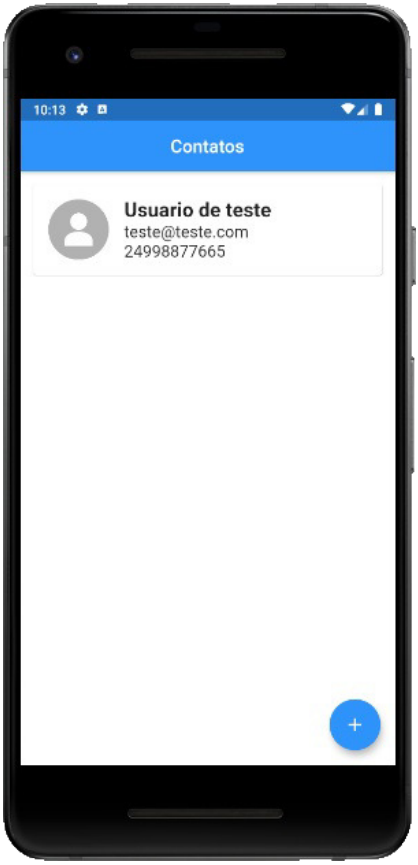

Figura 11.2: Listar contatos

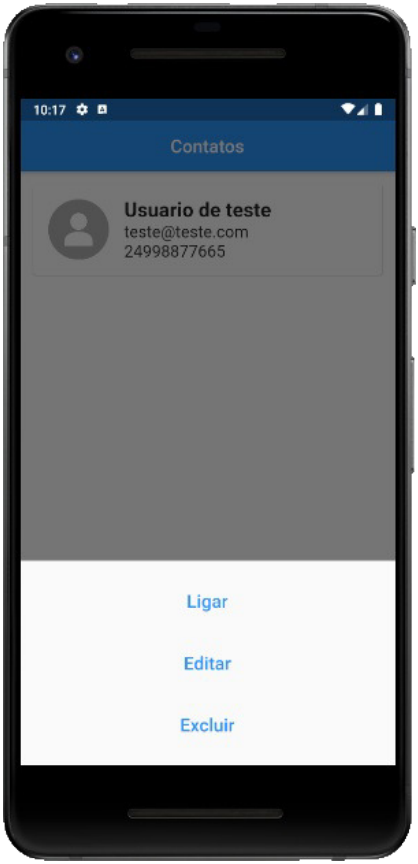

Figura 11.3: Menu de um contato específico selecionado na lista

11.2 MÃO NA MASSA!

Após a criação ser realizada, adicione no arquivo `pubspec.yam` as seguintes linhas:

```
dependencies:
      sqflite: ^1.2.0
      url_launcher: ^5.0.2
```

A chamada do `sqflite` vai importar o SQLite para o projeto e todas as funcionalidades que precisarmos para manipular o banco de dados. Já a linha do `url_launcher` nos possibilitará utilizar o pacote `launcher` que dará ao nosso aplicativo o poder suficiente para criarmos um botão chamado "ligar", que, quando o usuário pressionar, o direcionará para a tela de discagem com o número do contato selecionado já preenchido lá.

Ao manipular o arquivo `pubspec` , deve-se sempre tomar cuidado com a tabulação, afinal, a hierarquia dos elementos é definida através dela. Feito isso e salvo, seu editor de texto caso esteja configurado corretamente com o Dart vai baixar as dependências. Caso não esteja, através da linha de comando no diretório em que o seu projeto está execute o comando:

```
flutter packages get
```

E as dependências do `sqflite` e o `url_launcher` serão instaladas no projeto.

Por este aplicativo ser um pouco mais sofisticado estruturalmente falando, no mesmo diretório do arquivo `main.dart` crie uma pasta chamada `providers` e outra chamada `pages` . Dentro da pasta primeira, crie um arquivo chamado `database_provider.dart` . Já dentro da pasta `pages` crie dois arquivos, um será chamado de `home_page.dart` e o outro, de `contact_page.dart` .

O `database_provider.dart` é o arquivo que será responsável pela conexão com o banco de dados, assim como as funções de cadastrar, editar, remover e afins. Os arquivos `home_page.dart` e `contact_page.dart` ficarão encarregados de guardar o código das duas páginas que teremos no aplicativo.

Também utilizaremos um ícone com um bonequinho para simbolizar uma foto de perfil, então, crie uma pasta na raiz do projeto chamada `assets` e, dentro dela, outra, chamada `images`. Dentro desta última, insira a imagem que se encontra em: https://github.com/Leomhl/flutterbook_agenda/blob/master/assets/images/person.png. No final do processo, sua estrutura de pastas deverá ficar igual a da imagem:

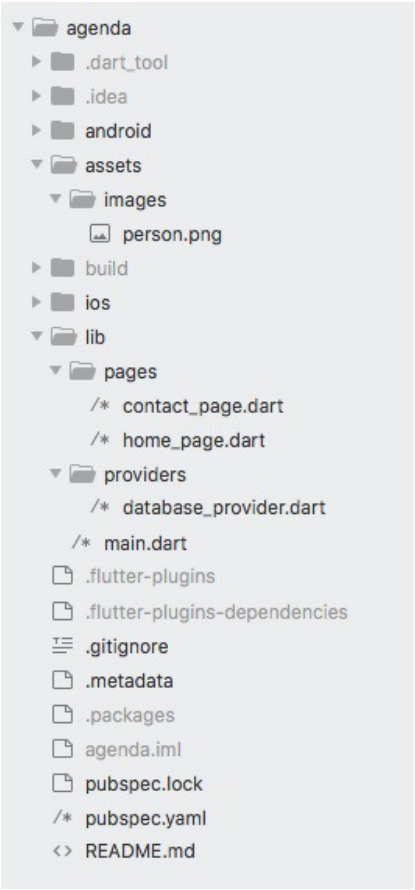

Figura 11.4: Diretório de arquivos do app agenda

Repare que removi a pasta `test` do meu projeto. Como não vamos realizar testes neste projeto aqui, por ser um recurso mais avançado que veremos mais para a frente, recomendo que você exclua esta pasta também para evitar possíveis erros futuros totalmente desnecessários.

Para o início do "coding", acesse o arquivo `main.dart` e apague todo o seu conteúdo. Na estrutura que criamos de `provider` e `page`, utilizaremos o `main` apenas para dar a "ignição" no aplicativo e o restante das responsabilidades ficarão com as páginas. Após limpar o arquivo `main.dart`, digite dentro dele o seguinte código:

```
import 'package:agenda/pages/home_page.dart';
import 'package:flutter/material.dart';

void main() {
  runApp(MaterialApp(
    home: HomePage(),
    debugShowCheckedModeBanner: false,
  ));
}
```

O arquivo `main.dart` terá apenas esse código. Nele, chamamos a tela Home e desabilitamos aquela tarja vermelha que fica ao topo do lado direito escrito "Debug" através da propriedade `debugShowCheckedModeBanner`.

Partindo agora para o provider de banco de dados, criaremos duas classes dentro dele. Uma que representa o contato e a outra que será o provider propriamente dito. Caso sinta-se confortável com Orientação a Objetos e classes e queira dividir o `database_provider.dart` em dois arquivos distintos, sem problemas! Trabalharemos aqui com apenas um para reduzir a complexidade do código e evitar que o código fique muito

espalhado, afinal, a ideia deste exemplo é ser o mais didático possível.

Dentro do arquivo `database_provider.dart` importaremos o `sqflite` e criaremos as classes `DatabaseProvider` e `Contact`. A classe `DatabaseProvider` cuidará especificamente das operações diretas ao banco de dados (criar, editar, remover, abrir e fechar conexão) enquanto a classe `Contact` focará em representar a estrutura de dados do contato, estrutura tal que a `DatabaseProvider` utilizará para salvar no banco de dados as informações que forem cadastradas, editadas ou removidas pelo usuário.

O arquivo `database_provider.dart` inicialmente ficará assim:

```
import 'package:sqflite/sqflite.dart';

class DatabaseProvider {
}

class Contact {
}
```

Precisamos definir quais serão os nomes dos campos na nossa tabela contatos do banco de dados. Como esses dados não serão alterados já que são valores estruturais vamos defini-los como constantes com a palavra reservada `final`. O arquivo ficará assim:

```
import 'package:sqflite/sqflite.dart';

// Nome dos campos na tabela do banco de dados, por isso são
constantes
final String contactTable = "contactTable";
final String idColumn = "idColumn";
final String nameColumn = "nameColumn";
```

```
final String emailColumn = "emailColumn";
final String phoneColumn = "phoneColumn";


class DatabaseProvider {
}

class Contact {
}
```

Agora, precisamos definir os parâmetros com que o usuário vai trabalhar, sendo eles id (identificador único numérico), nome, email e telefone. Após isso chamaremos o construtor da classe e posteriormente criaremos um segundo construtor que terá o papel de converter os dados de mapa (JSON) para objeto do contato. A classe `Contact` deverá ficar assim:

```
class Contact {
  int id;
  String name;
  String email;
  String phone;

  Contact();

  // Construtor que converte os dados de mapa (JSON) para obj
eto do contato
  Contact.fromMap(Map map){
    id = map[idColumn];
    name = map[nameColumn];
    email = map[emailColumn];
    phone = map[phoneColumn];
  }
}
```

Da mesma forma que convertemos os dados de um Mapa para um Objeto da classe `Contact`, também precisamos fazer o inverso para salvar os dados em formato de Mapa (JSON) no banco de dados. Para isso, criaremos um método chamado `toMap()`. A seguir, o código na íntegra da classe `Contact`

pronta!

```
    class Contact {

      int id;
      String name;
      String email;
      String phone;

      Contact();

      // Construtor que converte os dados de mapa (JSON) para obj
eto do contato
      Contact.fromMap(Map map){
        id = map[idColumn];
        name = map[nameColumn];
        email = map[emailColumn];
        phone = map[phoneColumn];
      }

      // Método que transforma o objeto do contato em Mapa (JSON)
 para armazenar no banco de dados
      Map toMap() {
        Map<String, dynamic> map = {
          nameColumn: name,
          emailColumn: email,
          phoneColumn: phone,
        };

        // O id pode ser nulo caso o registro esteja sendo criado
 já que é o banco de dados que
        // atribui o ID ao registro no ato de salvar. Por isso de
vemos testar antes de atribuir
        if(id != null){
          map[idColumn] = id;
        }
        return map;
      }
    }
```

11.3 SINGLETON

Para a classe `DatabaseProvider` utilizaremos um padrão diferente de construção. Como a ideia não é criar várias instâncias desta classe, afinal, temos apenas um banco de dados, utilizaremos o padrão **Singleton**.

Singleton é conhecido na verdade como uma espécie de antipadrão. Ele garante que exista apenas uma única instância de uma classe, mantendo um ponto acesso global ao seu objeto. Com isso, mesmo se criarmos novas instâncias da classe em diversos lugares do aplicativo, o objeto gerado pelo construtor será sempre o mesmo, logo, não teremos várias cópias em memória da classe. Trabalharemos sempre com uma única cópia da classe em que todos os objetos que foram criados dela irão apontar. Isso garante que trabalharemos com a instância do banco que abrimos, elimina redundâncias e traz uma excelente performance na comunicação com o banco de dados.

Existem diversas maneiras de criar Singletons em Dart. Utilizaremos a que julgo ser uma das mais simples de implementar, que utiliza apenas uma constante estática e uma factory para gerar uma instância fixa da classe através do construtor `internal` de que a linguagem Dart dispõe. Veja a seguir o código da implementação do construtor que gera o Singleton para a classe:

```
    class DatabaseProvider {

      static final DatabaseProvider _instance = DatabaseProvider.
internal();
      factory DatabaseProvider() => _instance;
      DatabaseProvider.internal();
```

```
}
```

Feito isso, precisamos declarar o objeto que será utilizado pela classe para se comunicar com o banco de dados. A partir de agora, trabalharemos com os tipos e recursos vindos do `sqflite`. Como a ideia é que apenas a classe `DatabaseProvider` comunique-se com o banco de dados, criaremos um atributo privado na classe, chamado `db`, que será do tipo Database. Para declarar um atributo privado devemos utilizar o underline. Então, insira abaixo da criação do construtor Singleton a seguinte declaração:

```
Database _db;
```

Para que o banco de dados seja inicializado a classe `DatabaseProvider`, é necessário um método dedicado a isso, então, criaremos um método que testa se o banco de dados já foi inicializado ou não. Caso não senha sido, invocará um outro método chamado `initDB` para realizar esta ação. A seguir, temos os códigos que realizam esta ação e que precisam ser posicionados abaixo da declaração do objeto `_db` que realizamos há pouco.

```
Future<Database> get db async {
    if(_db != null){
      return _db;
    } else {
      _db = await initDb();
      return _db;
    }
  }

  Future<Database> initDb() async {
    final databasesPath = await getDatabasesPath();
    final path = join(databasesPath, "contactsnew.db");

    return await openDatabase(path, version: 1, onCreate: (Da
tabase db, int newerVersion) async {
      await db.execute(
        "CREATE TABLE $contactTable($idColumn INTEGER PRIMARY
```

```
KEY, $nameColumn TEXT, $emailColumn TEXT,"
                "$phoneColumn TEXT)"
          );
        });
      }
```

O método `initDb` obtém o caminho do banco de dados que a biblioteca do `sqflite` gerou e concatena com o path do nosso aplicativo, assim, internamente o código saberá exatamente onde encontrar o banco. Após saber o caminho do banco de dados, invocamos o `openDatabase` , que fará o papel de criar a tabela que utilizaremos para salvar os dados, com o respectivo nome de cada campo que definimos lá no topo do arquivo em formato de constante. Repare que "path" não é um recurso nativo do Flutter assim como o assincronismo utilizado nas funções acima, então, declare na primeira linha do arquivo `database_provider.dart` as seguintes linhas:

```
import 'dart:async';
import 'package:path/path.dart';
```

É válido ressaltar que todas as funções que utilizamos são assíncronas, afinal, não sabemos quanto tempo exatamente levará até que a criação do banco, abertura da conexão e criação da tabela ocorram. Para não bloquear os demais processamentos que o aplicativo exige, utilizamos assincronismo. É através dele que as coisas respeitam uma ordem lógica de execução sem necessariamente sabermos quanto tempo levará para todo o ciclo de código ser executado.

O primeiro método que criaremos será o de salvar, afinal, de que adianta termos um banco de dados se não o utilizarmos para nada? O método `saveContact` receberá uma instância da classe `Contact` com todos os atributos que precisamos preenchidos. Ao

salvar no banco de dados, ele retornará o id daquele registro. Vamos inserir no objeto `contact` o id dele e retornar este objeto para o uso futuro nas nossas telas. Veja como criar o método para salvar um contato:

```
    Future<Contact> saveContact(Contact contact) async {
        Database dbContact = await db;
        contact.id = await dbContact.insert(contactTable, contact
.toMap());
        return contact;
    }
```

Repare que o método `saveContact` cria uma "cópia" para si do objeto `db` e executa uma operação de inserção passando como parâmetros a tabela que deve receber os dados e os dados propriamente ditos tratados pelo método `toMap()` que criamos anteriormente para formatar os dados para o formato de Mapa.

Feito o método para salvar um contato, agora precisamos de um que busque um contato específico, afinal, caso queiramos editar os dados de algum contato precisaremos desta funcionalidade. Basicamente, criaremos um método que recebe como parâmetro o id do contato e realiza a query no banco de dados solicitando todos os campos com informação daquele usuário cujo id estamos filtrando.

Para isso, precisaremos passar ao método `query` do `sqflite` a tabela em que gostaríamos de realizar a consulta, quais colunas queremos que a consulta retorne, uma cláusula `where` filtrando de qual usuário exatamente queremos os dados e, por último, qual é o id que será utilizado pelo `where` . Nosso método também utilizará o método `fromMap()` que criamos na classe `Contact` para fazer o caminho inverso do método de salvar. Agora tornaremos um Mapa em um objeto da classe `Contact` .

Veja como criar este método:

```
Future<Contact> getContact(int id) async {
  Database dbContact = await db;
  List<Map> maps = await dbContact.query(contactTable,
    columns: [idColumn, nameColumn, emailColumn, phoneColumn],
    where: "$idColumn = ?",
    whereArgs: [id]);

  if(maps.length > 0){
    return Contact.fromMap(maps.first);
  } else {
    return null;
  }
}
```

O método verifica se algum resultado foi obtido da busca pelo identificador do contato, caso nada tenha sido encontrado, ele retorna `null` .

Após a criação do método que busca um contato específico, agora criaremos um que remove um usuário específico. Este método será utilizado quando o usuário acionar o botão excluir presente no menu que abrirá ao selecionar um contato na lista de contatos cadastrados. Os comandos para exclusão são demasiadamente simples em comparação com os de filtragem e retorno de um dado. Para o método de exclusão basta passarmos o identificador do registro que pretendemos excluir e realizar uma query com a cláusula `where` filtrando pelo id do contato. Veja a implementação do método `deleteContact` .

```
Future<int> deleteContact(int id) async {
  Database dbContact = await db;
  return await dbContact.delete(contactTable, where: "$idColumn
= ?", whereArgs: [id]);
}
```

Além dos métodos criados anteriormente, também é

interessante que possamos dar ao usuário o poder de editar um contato, afinal, não faz sentido ele precisar remover e cadastrar novamente por culpa de algum pequeno erro pontual que pode ser facilmente corrigido. Para atualizar um contato, passaremos como parâmetro para o nosso método um objeto do tipo Contact com os dados do contato que queremos atualizar, assim, o método saberá em qual contato ele deve inserir quais dados.

```
  Future<int> updateContact(Contact contact) async {
    Database dbContact = await db;
    return await dbContact.update(contactTable,
        contact.toMap(),
        where: "$idColumn = ?",
        whereArgs: [contact.id]);
  }
```

E por último, mas não menos importante, precisaremos de um método que retorne todos os contatos para que sejam exibidos na tela principal do aplicativo. Este método realizará a seleção de todos os contatos que estão presentes na tabela e retornará uma lista com cada registro formatado para o formato de objeto da classe `Contact` .

```
  Future<List> getAllContacts() async {
    Database dbContact = await db;
    List listMap = await dbContact.rawQuery("SELECT * FROM $conta
ctTable");
    List<Contact> listContact = List();
    for(Map m in listMap){
      listContact.add(Contact.fromMap(m));
    }
    return listContact;
  }
```

Após estas etapas, o nosso provider de banco de dados está pronto! Agora precisamos fazer todo esse código funcionar em harmonia com as páginas que criaremos.

Para ver o provider na íntegra, o código está disponível no seguinte endereço:

https://github.com/Leomhl/flutterbook_agenda/blob/master/lib/providers/database_provider.dart

11.4 NEM SÓ DE PROVIDERS VIVE UM APP

As duas telas que criaremos terão funções bastante específicas. A tela principal (Home) será a primeira que o usuário verá ao abrir o aplicativo. Através de um botão, ela dará a opção de cadastrar um novo contato abrindo a tela de Contato. A tela principal também terá a função de listar os contatos existentes; ao tocar no cartão com os dados do usuário, a tela de contato será aberta com os dados daquele usuário, dados tais que estarão ali prontos para serem editados. Apesar de conceitualmente ambas as telas parecerem bastante simples, existe um grau de complexidade para tornar todo este mecanismo funcional. Este não é dos aplicativos mais difíceis de se criar, mas, para um iniciante em Flutter pode sim ter seu grau de complexidade a partir de agora. Então, mantenha a atenção redobrada e qualquer problema que ocorrer acesse o endereço com o código do projeto feito para este livro e compare com os seus códigos. Também é interessante manter-se atento à versão das extensões que estamos utilizando aqui. Atualizações futuras podem comprometer o nome dos métodos que estão sendo utilizados neste exemplo.

Para iniciarmos a implementação no arquivo `home_page.dart` , insira nas primeiras linhas os imports que precisaremos para trabalhar nesta tela. Ela utilizará o *url launcher* para abrir o menu de discagem do dispositivo do usuário e

preencher com o número do contato que estiver sendo manipulado naquele instante. Precisaremos da página de contato para direcionarmos a ela caso o usuário precise cadastrar ou editar algum contato, também precisaremos do provider de banco de dados para realizarmos consultas ao banco e trazer dados para a tela principal e, por último, precisaremos da biblioteca material para obtermos os Widgets necessários para a construção da tela. O início do seu arquivo `home_page.dart` deverá ficar assim:

```
import 'package:flutter/material.dart';
import 'package:agenda/pages/contact_page.dart';
import 'package:url_launcher/url_launcher.dart';
import 'package:agenda/providers/database_provider.dart';
```

A página `Home` demandará duas classes, sendo a principal, que será acessada de qualquer lugar da aplicação, e outra privada, que será acessada somente pela classe `Home` . A classe privada será o estado da classe `Home` que estende a classe `StatefulWidget` . Nesta página construiremos diversos métodos para nos auxiliarem na construção das funcionalidades da tela. Veremos adiante cada um deles. A classe principal, que deve ser declarada após os imports, é a seguinte:

```
class HomePage extends StatefulWidget {
  @override
  _HomePageState createState() => _HomePageState();
}
```

A segunda classe, declararemos da seguinte maneira:

```
class _HomePageState extends State<HomePage> {

}
```

Todo o restante do código da página será construído dentro do `_HomePageState` , afinal, é dentro do estado do Widget "pai" que

conseguiremos realizar as alterações de telas, textos e afins de que precisamos para tornar a tela dinâmica. De início, vamos declarar a variável que receberá uma instância Singleton da classe `DatabaseProvider`, e uma lista chamada `contacts`, que futuramente receberá todos os contatos cadastrados no banco de dados para que possam ser formatados e exibidos na tela. Seu código deverá ficar assim:

```
class _HomePageState extends State<HomePage> {

    DatabaseProvider database = DatabaseProvider();
    List<Contact> contacts = List();
  }
```

Para o próximo passo, sobrescreveremos o método `initState` da classe `State` para adaptá-lo à nossa necessidade. Precisamos que, quando o estado seja iniciado junto com a tela, ele faça a busca de todos os contatos cadastrados e carregue estes dados na lista de `contacts`. Abaixo da declaração da lista, crie o método `initState` da seguinte forma:

```
@override
void initState() {
  super.initState();

  _getAllContacts();
}
```

Você talvez reparou que chamamos um método privado da classe `Home` que ainda não existe, o `getAllContacts`. Este método é o responsável por solicitar ao `provider` do banco de dados a realização da consulta que retorna todos os contatos cadastrados. A implementação dele é bastante simples tendo em vista que "envelopamos" toda a lógica mais complexa dentro do `provider`. Agora basicamente acionaremos o `provider` para

trazer os dados e os passaremos para a lista, avisando ao estado para que ele atualize, e exiba as alterações realizadas na tela principal.

```
void _getAllContacts(){
  database.getAllContacts().then((list){
    setState(() {
      contacts = list;
    });
  });
}
```

Após a busca dos dados e atualização do estado, precisamos construir o Widget build que dará início a toda a interface gráfica da tela. É no método `build` que definiremos as propriedades da barra superior (`appBar`) como o título, cor de fundo e posicionamento do título. Também é nele que definiremos a cor de fundo da página, o botão flutuante que ficará posicionado no canto inferior direito da tela para que o usuário possa cadastrar um novo contato e, por último, o conteúdo da página com um componente chamado `ListView` . A seguir está o código do `build` :

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text("Contatos"),
      backgroundColor: Colors.blue,
      centerTitle: true,
    ),
    backgroundColor: Colors.white,
    floatingActionButton: FloatingActionButton(
        onPressed: (){
          _showContactPage();
        },
        child: Icon(Icons.add),
        backgroundColor: Colors.blue,
     ),
    body: ListView.builder(
```

```
          padding: EdgeInsets.all(10.0),
          itemCount: contacts.length,
          itemBuilder: (context, index) {
            return _contactCard(context, index);
          }
      ),
    );
  }
```

Ao analisar o código, você deve ter percebido que ainda não há na classe algumas coisas que chamamos dentro do método, como o `contactCard` e o `showContactPage`. O `contactCard` é o Widget responsável pela exibição gráfica dos dados dos contatos, enquanto o `showContactPage` é o método que realiza a chamada para a abertura da tela de Contato.

Criaremos agora o Widget `contactCard`, onde teremos um Widget chamado `GestureDetector` que ficará "vigiando" qualquer movimento que o usuário faça na tela. Caso o usuário dê um tap (click) em algum elemento Card que estiver dentro dele, ou seja, em algum contato, ele prontamente disparará a chamada do método `showOptions`, que abrirá um menu de opções. Criaremos o método `showOptions` posterior a este. Basicamente o nosso card terá uma linha com duas colunas, sendo uma para a imagem de perfil e outra para os dados do usuário. Veja o código do Widget `contactCard`.

```
Widget _contactCard(BuildContext context, int index){
    return GestureDetector(
      child: Card(
        child: Padding(
            padding: EdgeInsets.all(10.0),
            child: Row(
              children: <Widget>[
                Container(
                  width: 80.0,
                  height: 80.0,
```

```
                    decoration: BoxDecoration(
                      shape: BoxShape.circle,
                      image: DecorationImage(
                        image: AssetImage("assets/images/person
.png"),
                        fit: BoxFit.cover
                      ),
                    ),
                  ),
                  Padding(
                    padding: EdgeInsets.only(left: 10.0),
                    child: Column(
                      crossAxisAlignment: CrossAxisAlignment.st
art,
                      children: <Widget>[
                        Text(contacts[index].name ?? "",
                          style: TextStyle(fontSize: 22.0,
                              fontWeight: FontWeight.bold),
                        ),
                        Text(contacts[index].email ?? "",
                          style: TextStyle(fontSize: 18.0),
                        ),
                        Text(contacts[index].phone ?? "",
                          style: TextStyle(fontSize: 18.0),
                        )
                      ],
                    ),
                  )
                ],
              ),
            ),
          ),
          onTap: (){
            _showOptions(context, index);
          },
        );
    }
```

O método `showOptions` é o responsável por exibir o menu com as opções de ligar, editar e excluir o contato. Para evitar repetirmos três vezes o mesmo código, um para cada botão, criaremos um Widget personalizado posteriormente chamado

`optionButton` e passaremos para ele o título e a ação que vai realizar. Assim utilizaremos a componentização a nosso favor para economizar linhas de código e organizar o projeto de forma mais profissional. Veja como ficará o método `showOptions`:

```
  void _showOptions(BuildContext context, int index){
    showModalBottomSheet(
        context: context,
        builder: (context){
          return BottomSheet(
            onClosing: (){},
            builder: (context){
              return Container(
                padding: EdgeInsets.all(10.0),
                child: Column(
                  mainAxisSize: MainAxisSize.min,
                  children: <Widget>[

                    _optionButton(context, "Ligar", (){
                        launch("tel:${contacts[index].phone}");
                        Navigator.pop(context);
                      },
                    ),

                    _optionButton(context, "Editar", (){
                        Navigator.pop(context);
                        _showContactPage(contact: contacts[index]
);
                      },
                    ),

                    _optionButton(context, "Excluir", (){
                        database.deleteContact(contacts[index].id
);
                        setState(() {
                          contacts.removeAt(index);
                          Navigator.pop(context);
                        });
                      },
                    ),
                  ],
                ),
              );
```

```
            },
          );
        }
    );
  }
```

Para concluir o código do método `showOptions`, agora criaremos nosso Widget personalizado `optionButton`. Veja como deverá ficar o código:

```
  Widget _optionButton(BuildContext context, title, pressedFuncti
on) {
    return Padding(
            padding: EdgeInsets.all(10.0),
            child: FlatButton(
              child: Text(title,
                style: TextStyle(color: Colors.blue, fontSize: 20
.0),
              ),
              onPressed: pressedFunction
            ),
          );
  }
```

Por último, criaremos o método `showContactPage`, que tem a função de invocar a tela de Contato passando para ela os dados do usuário selecionado caso ele tenha acionado a edição, ou a abre de forma não preenchida para o cadastro de um novo usuário ser realizado. Veja a implementação deste método.

```
  void _showContactPage({Contact contact}) async {
    final recContact = await Navigator.push(context,
      MaterialPageRoute(builder: (context) => ContactPage(contact
: contact,))
    );
    if(recContact != null){
      if(contact != null){
        await database.updateContact(recContact);
      } else {
        await database.saveContact(recContact);
      }
```

```
      _getAllContacts();
    }
  }
```

Para ver a tela home na íntegra, o código está disponível no seguinte endereço:

https://github.com/Leomhl/flutterbook_agenda/blob/master/lib/pages/home_page.dart

Já construímos o provider de banco de dados e a tela principal Home, agora, precisamos construir a tela de Contato para findar nosso aplicativo. A tela de contato é a tela que se responsabilizará pelas operações mais importantes, a de cadastrar e editar um contato. Digo isso pois sem essas funcionalidades a regra de negócio do nosso aplicativo não funcionaria, deixando-o sem serventia prática. Mãos à obra! Abra o arquivo `contact_page.dart` dentro da pasta `pages`. A primeira coisa que faremos nele serão os imports das bibliotecas que utilizaremos para implementar a página. Insira no seu arquivo os seguintes imports:

```
import 'dart:async';
import 'package:flutter/material.dart';
import 'package:agenda/providers/database_provider.dart';
```

Agora, criaremos a classe `ContactPage`, que terá uma constante chamada `contact`. Esta constante terá seu valor atribuído quando a tela abrir: caso seja uma operação de edição que abrir a tela, os valores desta constante serão os dados do contato que está sendo editado. Caso seja uma operação de cadastro que abriu a tela, automaticamente o valor do objeto será vazio. Veja como a classe `ContactPage` deve ficar:

```
class ContactPage extends StatefulWidget {
```

```
  final Contact contact;

  ContactPage({this.contact});

  @override
  _ContactPageState createState() => _ContactPageState();
}
```

Criada a classe, agora precisaremos implementar o estado dela assim como fizemos na classe `HomePage`. Para isso utilizaremos as seguintes linhas de código:

```
class _ContactPageState extends State<ContactPage> {
}
```

Dentro do estado criaremos três controllers. Os controllers serão responsáveis pelos campos de texto, sendo eles: nome, email e telefone. São eles que gerenciarão as alterações feitas pelo usuário. Utilizaremos constantes final para os controllers já que eles não devem ser alterados após criados; caso sejam alterados perderemos o controle dos campos que temos na tela. Para isso, crie as seguintes linhas dentro do `ContactPageState`.

```
final _nameController = TextEditingController();
final _emailController = TextEditingController();
final _phoneController = TextEditingController();
```

Agora criaremos três elementos, sendo eles uma constante e duas variáveis. A constante será responsável por controlar o autofoco no campo de nome caso o usuário do aplicativo tente salvar um contato sem sequer informar um nome. A segunda variável controla se um usuário está sendo editado naquele momento, caso esteja através dessa variável saberemos se tem dados não salvos na tela, e assim conseguiremos enviar uma mensagem ao usuário de que ele perderá as alterações caso não

salve os valores da tela antes de sair dela. E por último, a variável que armazena os dados do usuário que está sendo editado pelo estado da aplicação. Veja a seguir a declaração destes três itens:

```
final _nameFocus = FocusNode();

bool _userEdited = false;

Contact _editedContact;
```

Prosseguindo, precisaremos sobrescrever o método `initState` da classe `State` assim como fizemos no arquivo `home_page.dart`. É através do `init` que testaremos se a tela foi chamada em modo de edição ou de cadastro; caso seja de edição, ela carrega nos controllers os valores vindos do banco de dados. Veja o código da sobrescrita do método `initState`:

```
@override
void initState() {
  super.initState();

  if(widget.contact == null){
    _editedContact = Contact();
  } else {
    _editedContact = Contact.fromMap(widget.contact.toMap());

    _nameController.text = _editedContact.name;
    _emailController.text = _editedContact.email;
    _phoneController.text = _editedContact.phone;
  }
}
```

Para o método `build` da página, vamos iniciá-lo utilizando um Widget bastante útil, chamado `WillPopScope`. Este Widget alertará quando a página receber o comando de sair, e, caso os dados editados não tenham sido salvos, este Widget se encarregará de executar uma função que criaremos especialmente para alertar ao usuário da possível perda, caso realmente queira sair da página.

Como elemento filho do `WillPopScope`, utilizaremos o `Scaffold`, que é amplamente conhecido e utilizado para a criação de telas como as que fizemos anteriormente. No `Scaffold` da tela de Contatos faremos um tratamento para que, caso a tela esteja sendo usada para edição, exiba como título o nome do contato. Caso contrário, ela exibirá o texto genérico "Novo Contato".

Logo após a parte da appBar, configuraremos no `Scaffold` o botão flutuante que testará se os dados podem ser salvos ou se nada foi informado no campo de nome. Admito que é uma validação bastante básica, mas tem o que é necessário para entender o processo de validação. Caso queira melhorar o algoritmo e criar uma validação completa, sinta-se livre! Você é totalmente capaz para isso, fica como desafio.

E por último, configuraremos o body, que tem a responsabilidade de construir os elementos do corpo da página. Nele, utilizaremos um Widget `Column` para servir de espaço para a imagem do avatar de perfil que estará dentro de um outro Widget chamado `Container`, e para os campos de texto que o usuário informará os dados do contato. Veja como deverá ser feita a implementação do método `build`:

```
@override
Widget build(BuildContext context) {
  return WillPopScope(
    onWillPop: _requestPop,
    child: Scaffold(
      appBar: AppBar(
        backgroundColor: Colors.blue,
        title: Text(_editedContact.name ?? "Novo Contato"),
        centerTitle: true,
      ),
      floatingActionButton: FloatingActionButton(
```

```
          onPressed: (){
            if(_editedContact.name != null && _editedContact.name
.isNotEmpty){
              Navigator.pop(context, _editedContact);
            } else {
              FocusScope.of(context).requestFocus(_nameFocus);
            }
          },
          child: Icon(Icons.save),
          backgroundColor: Colors.blue,
        ),
        body: SingleChildScrollView(
          padding: EdgeInsets.all(10.0),
          child: Column(
            children: <Widget>[
              Container(
                width: 140.0,
                height: 140.0,
                decoration: BoxDecoration(
                  shape: BoxShape.circle,
                  image: DecorationImage(
                      image: AssetImage("assets/images/person.png
),
                      fit: BoxFit.cover
                  ),
                ),
              ),
              TextField(
                controller: _nameController,
                focusNode: _nameFocus,
                decoration: InputDecoration(labelText: "Nome"),
                onChanged: (text){
                  _userEdited = true;
                  setState(() {
                    _editedContact.name = text;
                  });
                },
              ),
              TextField(
                controller: _emailController,
                decoration: InputDecoration(labelText: "Email"),
                onChanged: (text){
                  _userEdited = true;
                  _editedContact.email = text;
                },
```

```
                keyboardType: TextInputType.emailAddress,
              ),
              TextField(
                controller: _phoneController,
                decoration: InputDecoration(labelText: "Phone"),
                onChanged: (text){
                  _userEdited = true;
                  _editedContact.phone = text;
                },
                keyboardType: TextInputType.phone,
              ),
            ],
          ),
        ),
      ),
    );
  }
```

Para finalizar, precisamos criar o método `requestPop`, que é chamado no Widget `WillPopScope`. Este método verificará se o atributo privado `userEdited` é verdadeiro. Caso seja, o usuário realizou alterações nos valores dos campos que estão carregados com os dados de um contato específico. Caso o atributo `userEdited` esteja como verdadeiro, ele criará uma janela de diálogo comunicando o usuário de que, se ele sair da tela sem salvar as alterações, vai perdê-las, e dá a opção de sair ou continuar na tela. Veja o código deste método na íntegra:

```
  Future<bool> _requestPop(){
    if(_userEdited){
      showDialog(context: context,
        builder: (context){
          return AlertDialog(
            title: Text("Descartar Alterações?"),
            content: Text("Se sair as alterações serão perdidas."
),
            actions: <Widget>[
              FlatButton(
                child: Text("Cancelar"),
                onPressed: (){
                  Navigator.pop(context);
```

```
              },
            ),
            FlatButton(
              child: Text("Sim"),
              onPressed: (){
                Navigator.pop(context);
                Navigator.pop(context);
              },
            ),
          ],
        );
      }
    );
    return Future.value(false);
  } else {
    return Future.value(true);
  }
}
```

Repare que no `onPressed` temos uma duplicidade na linha `Navigator.pop(context);` . Essa duplicidade não é falha de programação, é proposital. O primeiro vai fechar a janela de diálogo que foi aberta para perguntar ao usuário se ele deseja prosseguir. A segunda fechará a tela de contatos, assim, retorna para a página Home que é a primeira da pilha de páginas. Veja na figura a seguir como funciona a validação de sair da tela sem salvar os dados.

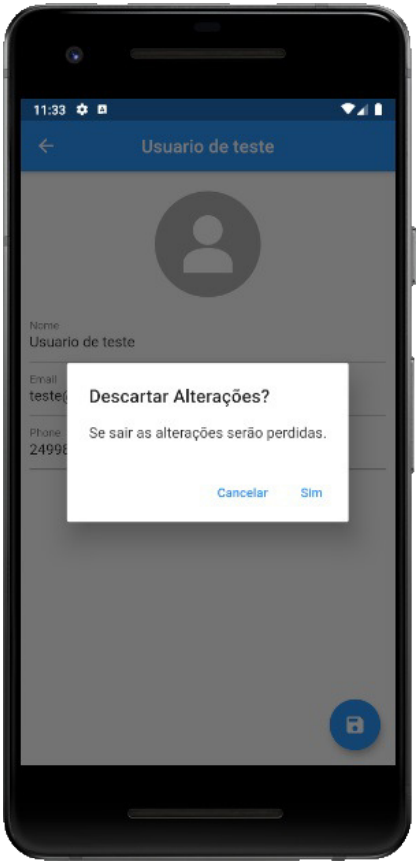

Figura 11.5: Validação de saída sem salvar

Pronto! Finalmente temos a criação completa. Espero que os conceitos de banco de dados e multitela tenham ficados claros. Nesse exemplo, simplicidade e objetividade foram os conceitos que empregamos para entender melhor a utilização de uma base de dados local em seu aplicativo. Utilizamos de forma simples e bem básica o banco de dados. Nos seus projetos você pode utilizar o banco para salvar dados enquanto não estiver conectado com a

internet e posteriormente sincronizar com um servidor por meio de API, por exemplo. As possibilidades são praticamente infinitas do que pode ser feito com esse recurso tão legal que aprendemos a explorar. Espero de verdade que seja bastante útil e que agregue positivamente aos seus projetos.

Para este projeto na íntegra, o código está disponível no seguinte endereço:

https://github.com/Leomhl/flutterbook_agenda

Capítulo 12

TESTES AUTOMATIZADOS EM WIDGETS

12.1 PARA QUE TESTAR?

Desde que o desenvolvimento de softwares começou a ser moldado lá pelos anos 1950, com os primeiros computadores que tinham o tamanho de um andar de prédio, buscam-se formas de tornar as instruções que eram dadas a eles as mais corretas possível. Um pequeno erro em algum comando acarretaria em resultados errados e consequentemente uma perda de tempo e dinheiro monumental para a época. Muito tempo se passou desde esta era, mas os princípios fundamentais do desenvolvimento de software se mantiveram. O erro humano ainda é bastante presente e comum na programação de máquinas, afinal, humanos erram e isso não é nenhum tipo de novidade.

Com o passar das décadas, pesquisadores e empresas foram desenvolvendo por meio de experimentação e empirismo táticas para testar os softwares e encontrar as possíveis falhas escondias. Dentro destas pesquisas, buscando uma forma mais precisa e simples de encontrar falhas, notaram que quando uma pessoa que não era da equipe de desenvolvimento testava o sistema, normalmente ela encontrava mais falhas do que quem havia

acompanhando o processo de criação do código. Com isso, ficou constatado que ver a criação do código gera uma espécie de "vício" na pessoa, afinal, ela sabe qual valor um campo numérico espera, por exemplo. Já quem não acompanhou o desenvolvimento talvez nem imagine que um determinado campo deva ser numérico e insere valores não esperados por quem o criou, logo, encontraremos problemas se as validações não forem feitas corretamente.

O grande problema dos testes manuais é que, caso uma pequena atualização seja feita em uma tela, por exemplo, o testador possivelmente não realizará longos testes novamente por já ter feito isso, ou, testará da mesma forma que fez anteriormente, o que deixa uma pequena - mas real - probabilidade de não contemplar algum cenário e deixar passar uma falha. Em resumo, quanto mais humanizado o processo de testes, maiores são as chances de problemas relativamente simples passarem despercebidos.

Visando resolver este impasse que se arrasta a quase um século, criou-se o conceito de **teste automatizado**. Os testes automatizados conseguem realizar com louvor e em um tempo insanamente mais rápido que um humano.

Partindo da criação de bibliotecas de testes automatizados, perceberam que seria interessante o início de uma nova cultura entre os programadores. A esta cultura chamaram de **TDD (Test-driven development)**. O TDD, em português, desenvolvimento guiado por testes, é uma filosofia de desenvolvimento na qual, conforme o programador vai criando as funcionalidades do sistema ele próprio cria testes que validam o resultado esperado de cada função, assim, qualquer microalteração que torne o resultado

diferente do esperado será acusada como falha. Utilizando TDD, o código muito provavelmente encontrará durante a execução dos testes um problema que talvez um testador humano não fosse encontrar. A vantagem disso é a economia de tempo, pessoas e aumento gigantesco na precisão dos testes já que eles podem ser executados dezenas de vezes ao dia testando toda a aplicação em pouquíssimos segundos.

Os testes automatizados resolveram a velha novela na vida dos programadores de subir atualizações na sexta-feira e perder o sábado trabalhando remoto para resolver as falhas enquanto os clientes ligam furiosos para os responsáveis pela empresa. Se rodar os testes na própria sexta-feira já será possível saber os erros que vão ocorrer e corrigi-los antes de o problema ocorrer em modo de produção com os clientes utilizando.

12.2 CRIANDO TESTES AUTOMATIZADOS COM DART

Criaremos aqui uma aplicação que calcula descontos para vermos na prática como gerar testes! Para isso, crie um aplicativo chamado "desconto".

Na pasta `tests`, remova o arquivo `widget_test.dart` e crie um novo chamado `desconto_test.dart`. Feito isso, esvazie o arquivo `main.dart` também. Vamos começar bem do zero para entender o processo. O Flutter traz para nós uma biblioteca de testes automatizados totalmente de graça, não precisamos instanciar alguma dependência específica para isso, pois ela já vem por padrão no `pubspec`. Verifique se no seu arquivo `pubspec.yam` vêm as seguintes linhas inclusas:

```
dev_dependencies:
  flutter_test:
    sdk: flutter
```

É interessante notar que a dependência de teste vem por padrão na parte de dependências de desenvolvimento (`dev_dependencies`), ou seja, quando o aplicativo é gerado para a loja de aplicativos em versão de produção, esta dependência não será instalada nele, logo, economizaremos um bom espaço que seria inutilizado dentro do executável que enviaremos para a loja.

Verificada a dependência de testes, precisamos entender a lógica por trás da criação de teste. Um teste é muito simples: quando uma pessoa vai testar algo, ela sabe o que quer testar e qual é o resultado esperado após uma determinada ação. Então, temos que ficar atentos a três coisas. A primeira é o que testar, por exemplo, "Deve dar erro ao calcular com valor inicial negativo". A segunda é a ação que ela vai realizar para obter o resultado do teste. Como estamos trabalhando no nível do código, é bem mais simples descrever passo a passo o teste por meio da linguagem Dart. Terceiro e último, precisamos saber qual é o resultado esperado que deve sair da execução da função após darmos uma determinada entrada de dados. E pronto, a lógica do teste está pronta. Basicamente programar testes é criar uma representação em linguagem de programação do que já era feito de forma manual e repetitiva humanamente.

Vamos começar no arquivo `test/desconto_test.dart` importando as seguintes linhas:

```
import 'package:flutter_test/flutter_test.dart';
import 'package:desconto/main.dart';
```

Agora criaremos o módulo `main` com uma constante

informando o valor sem desconto que utilizaremos para todo o funcionamento dos testes. Digite no seu arquivo de teste as seguintes linhas após os imports:

```
void main() {
      const valorSemDesconto = 150.0;
  }
```

Antes de criar a sintaxe do teste, é interessante que a função que se deseja testar já exista. Claro que você pode programar o teste antes de criar a função, mas para evitar que fique aparecendo erro na hora que você criar o teste para este exemplo optaremos por criar primeiro a função. Nossa função de desconto ficará no arquivo `main.dart` e será utilizada pelo aplicativo. Ela receberá três parâmetros, sendo eles o valor sem desconto, o valor do desconto e se é um desconto inteiro ou percentual. Dentro da função, validaremos se o desconto é maior do que zero, se o valor inicial é maior do que zero e se o desconto será realizado utilizando um cálculo inteiro ou percentual. Deixe o seu arquivo `main.dart` da seguinte forma:

```
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Desconto',
      home: MyHomePage(),
      debugShowCheckedModeBanner: false,
    );
  }
}

class MyHomePage extends StatefulWidget {
```

```
  @override
  _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Desconto"),
      ),
      body: Container(),
    );
  }
}

double calcularDesconto(double initValue, double discount, in
 percentual) {
  if (discount <= 0)
    throw new ArgumentError("O desconto deve ser maior que ze
ro!");

  if (initValue <= 0)
    throw new ArgumentError("O valor inicial deve ser maior q
ue zero!");

  if (percentual != 0)
    return initValue - (initValue * discount / 100);

  return initValue - discount;
}
```

Criada a função e preparativos iniciais para o funcionamento do aplicativo, podemos finalmente escrever nosso primeiro teste. Dentro do arquivo `desconto_test.dart`, abaixo da declaração da constante `valorSemDesconto`, escreva o seguinte código:

```
  test('Deve clacular desconto corretamente utilizando números de
cimais', () {
  const desconto = 25.0;
  const valorComDesconto = valorSemDesconto - desconto;
```

```
    expect(
        calcularDesconto(valorSemDesconto, desconto, 0), valorCom
Desconto);
  });
```

Basicamente o método de teste espera dois parâmetros: o primeiro é a descrição do teste e o segundo é uma função que vai executar a chamada do método que queremos testar. Repare que dentro desta função anônima informamos os valores que queremos inserir no método `calcularDesconto`, e o inserimos dentro de um método chamado `expect`. O `expect` necessita que sejam informados dois parâmetros: o primeiro é a função que estamos testando para que ele possa capturar o valor de retorno da sua execução, e o segundo é o que esperamos que a função retorne. Assim ele consegue reparar se o que foi retornado faz sentido e é o esperado; caso não seja, ele vai disparar um erro.

Para executar o teste é bastante simples, abra o seu terminal de comandos e navegue até a pasta do projeto. Dentro da pasta do projeto rode o seguinte comando no terminal:

```
flutter test test/widget_test.dart
```

Caso você queira criar diversos arquivos de teste, para executar todos de uma vez utilize este comando:

```
flutter test test/*
```

Veja o resultado da execução do teste:

```
MacBook-Pro-de-Leonardo:desconto leonardomarinho$ flutter test test/desconto_test.dart
00:03 +5: All tests passed!
MacBook-Pro-de-Leonardo:desconto leonardomarinho$
```

Figura 12.1: Execução dos testes

Agora, vamos criar mais testes para a função de calcular desconto. Precisamos saber se ela funciona corretamente calculando descontos via porcentagem, se a validação de o desconto ser negativo ou zero está funcionando, se o valor inicial é zero ou se o valor inicial é negativo. Para averiguar se o cálculo é feito corretamente com porcentagem escreva o seguinte teste:

```
    test('Deve calcular o desconto corretamente utilizando porcen
tagem', () {
        var desconto = 10.0;
        var valorComDesconto = 135.0;
        expect(calcularDesconto(valorSemDesconto, desconto, 1), v
alorComDesconto);
    });
```

Para verificarmos se funcionam corretamente as `exceptions` que construímos ao disparar um erro caso o valor com desconto seja zero ou negativo, é necessário um objeto especial chamado `throwsA` . Ele verifica se houve no código um disparo de `throw exception` e realiza o teste. Repare que, para testar as exceções, a sintaxe é um pouco diferente da convencional, precisamos chamar uma função anônima como primeiro parâmetro para executar a função de calcular desconto e como segundo parâmetro (resultado esperado) passamos um objeto que vem do tipo das exceções. Note que é possível criar vários `expects` dentro do mesmo teste, mas, caso um falhe, somente a mensagem definida na descrição do teste será exibida.

```
    test('Deve dar erro ao calcula valor com desconto negativo ou
 zero', () {
        expect(() => calcularDesconto(valorSemDesconto, -1, 1),
            throwsA(isA<ArgumentError>()));

        expect(() => calcularDesconto(valorSemDesconto, 0, 0),
            throwsA(isA<ArgumentError>()));
    });
```

```
    test('Deve dar erro ao calcular desconto com valor inicial ze
ro', () {
        expect(() => calcularDesconto(0, 15, 0),
            throwsA(isA<ArgumentError>()));
    });


    test('Deve dar erro ao calcular com valor inicial negativo',
() {
        expect(() => calcularDesconto(-1, 15, 0),
            throwsA(isA<ArgumentError>()));
    });
}
```

Para terminarmos a lógica do aplicativo tornando-o útil para um usuário comum, dentro da classe `_MyHomePageState` , declare os controllers dos campos que utilizaremos e a variável que armazenará o valor do resultado do cálculo de desconto.

```
  var initValueController = TextEditingController();
  var discountController = TextEditingController();
  var percentualController = TextEditingController();
  double result = 0;
```

Agora, precisamos inserir campos na tela. Utilizaremos um Widget `Text` para exibir o valor calculado com desconto, e três campos de texto para o usuário informar o valor inicial, o desconto, e se o desconto é percentual ou inteiro. O seu Widget `build` da classe `MyHomePageState` deverá ficar assim:

```
    @override
    Widget build(BuildContext context) {
        return Scaffold(
          appBar: AppBar(
            title: Text("Desconto"),
          ),
          body: Container(
            child: SingleChildScrollView(
            padding: EdgeInsets.all(10.0),
            child: Column(
              crossAxisAlignment: CrossAxisAlignment.stretch,
```

```
              children: <Widget>[
                Divider(),
                Text(
                  result.toString(),
                  textAlign: TextAlign.center,
                  style: TextStyle(fontWeight: FontWeight.bold, f
ontSize: 30),
                ),
                Divider(),
                buildTextField("Valor inicial", initValueControll
er),
                Divider(),
                buildTextField("Desconto", discountController),
                Divider(),
                buildTextField("Inteiro/decimal", percentualContr
oller),
                Divider(),
                RaisedButton(
                  onPressed: (){
                    setState(() {
                      result = calcularDesconto(double.parse(init
ValueController.text), double.parse(discountController.text), int
.tryParse(percentualController.text));
                    });
                  },
                  child: Text(
                    'Calcular',
                    style: TextStyle(fontSize: 20)
                  ),
                  textColor: Colors.white,
                  color: Colors.blue,
                  padding: const EdgeInsets.all(8.0),
                ),
              ],
            ),
            ),
          ),
        );
    }
```

Por último, fora da classe de estado, crie o método que montará o Widget de texto que o usuário utilizará.

```
    Widget buildTextField(String label, TextEditingController c){
```

```
      return TextField(
        controller: c,
        decoration: InputDecoration(
            labelText: label,
            labelStyle: TextStyle(color: Colors.blue),
            border: OutlineInputBorder(),
        ),
        style: TextStyle(
            color: Colors.blue, fontSize: 25.0
        ),
        keyboardType: TextInputType.numberWithOptions(decimal: tr
ue),
      );
    }
```

Seu aplicativo deverá ficar como a imagem a seguir:

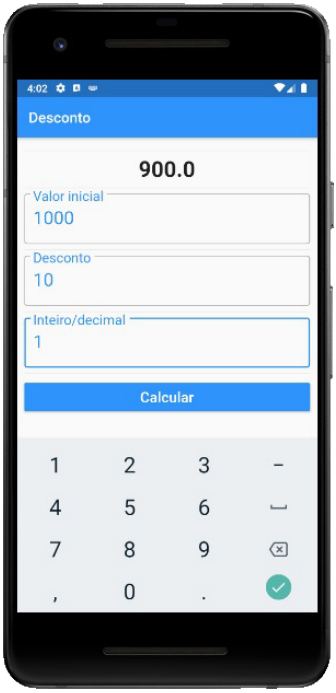

Figura 12.2: Aplicativo testado automaticamente

Desta forma, conseguimos criar testes para nossos aplicativos de maneira simples e eficiente para assegurar que tudo está correto e funcionando como esperado. O produto final é uma aplicação segura que o usuário poderá utilizar para calcular descontos sobre qualquer coisa. Ainda existem diversos comando para refinar os testes e simular ações humanas. Caso tenha interesse, investigue melhor a biblioteca de testes no Flutter na documentação oficial para encontrar as demais funcionalidades que podem ser utilizadas para resguardar o sistema de possíveis falhas.

Para ver o código deste projeto na íntegra acesse o seguinte endereço:

https://github.com/Leomhl/flutterbook_desconto

Capítulo 13

MUDANÇA DO ÍCONE DO APLICATIVO

Breve reflexão

Como em toda boa obra, precisamos nos preocupar com a criação de um sólido alicerce para manter a estrutura firme e segura. A parte estrutural de uma construção precisa ser feita com demasiado cuidado pela equipe técnica para que nada saia do esperado no projeto. Porém, de nada adianta uma construção extremamente forte e segura se esteticamente estiver em tijolos e cimento. Quando eu era adolescente, ouvi uma frase que nunca esqueci, "as pessoas compram com os olhos". Sempre achei engraçada pela redundância, afinal, ninguém tira os olhos para comprar algo. Mas o real sentido dela não está nesta piadinha infame e sim no ensinamento de que a estética é sim um dos maiores determinantes de sucesso para um bom produto.

Assim como uma casa precisa ser muito bem acabada e decorada para ter uma venda de sucesso, softwares não são diferentes. Os usuários compram pelo visual um aplicativo e só depois testam as funcionalidades. A preocupação com a criação de uma estética que represente a razão pela qual um aplicativo existe é fundamental para a permanência da aceitação de mercado. A

escolha do ícone que o aplicativo terá pode parecer uma escolha bastante irrisória para a maioria, mas é uma das mais importantes que você fará ao longo do ciclo de vida do app. Afinal, sempre que seu aplicativo estiver fechado e alguns dias sem ser usado pelo usuário, será o ícone o chamariz para que o usuário lembre do seu app e o abra.

Ao mesmo tempo que o ícone deve ser um bom chamariz, necessita representar a mensagem do seu negócio e fixar a marca na mente do usuário. Quanto mais inteligente for o design de um ícone mais será criado o efeito "coca-cola", você será o primeiro lembrado ao precisarem de serviços dos quais o seu aplicativo oferece. Então, ao criar um projeto não economize em design para custear apenas o desenvolvimento. Invista no desenvolvimento de uma marca única que represente seu negócio ou do seu cliente caso esteja criando uma aplicação para terceiros.

Partindo para a prática

Confesso que quando comecei minha longa jornada de estudos do Flutter encontrei um pouco de dificuldades para entender a lógica de gerar o ícone principal da aplicação. Conhecia a lógica de ícones do desenvolvimento para Android nativo, segundo a qual é necessário gerar mais de uma dezena de ícones nas mais variadas resoluções e cores de fundo para o projeto, o que sempre achei burocrático e um tanto chato de fazer. Quando migrei para o Cordova, poucos anos depois, ainda era estranho mas alguns desenvolvedores criaram scripts em JavaScript para redimensionar as imagens e realizar o trabalho de forma mais automatizada, o que - diga-se de passagem - foi excelente. Posteriormente, com a vinda do Ionic framework, foi inserido um comando quase divino

chamado `ionic resources`. Com ele, basta criar na raiz do projeto uma pasta chamada resources e inserir duas imagens com a extensão `.png`, uma para a imagem de `splash screen` e outra para o ícone. E pronto, o comando fazia absolutamente tudo, desde redimensionar até inserir nos diretórios corretos as imagens nas resoluções necessárias.

Quando vim para o Flutter, esperava encontrar no `CLI` (interface de linha de comando) algum comando que, assim como o Ionic, "automagicamente" gerasse os ícones. Não esperava encontrar algo focado em `splash screen` já que este conceito está caindo no meio de *UX mobile*. Executando o comando `flutter` no meu terminal veio a interrogação na mente. Não tem um comando específico para gerar ícone. E agora? Pesquisando sobre este tópico, logo entendi que a arquitetura era um pouco diferente daquela com que eu estava acostumado. Não duvido que em futuras atualizações do `CLI` do Flutter possa vir o comando para gerar o ícone do app, mas, até lá precisamos de um plugin que foi gentilmente desenvolvido e disponibilizado pela comunidade Flutter no repositório oficial da linguagem Dart.

Trabalharemos em cima deste plugin, e no final, nosso resultado será uma aplicação com um lindo ícone personalizado. Para isso, inicie um projeto chamado "icone". Não crie o projeto chamado "icon", pode haver incompatibilidades internas por ser uma palavra reservada por alguns Widgets e bibliotecas. Ao criar e executar o projeto em um emulador ou dispositivo físico, logo veremos que a nossa aplicação veio com o ícone padrão do Flutter. Veja como é este ícone:

Figura 13.1: Ícone padrão do Flutter

Para alterar este ícone utilizaremos um plugin chamado `flutter_launcher_icons` . Para saber mais sobre este pacote acesse: https://pub.dev/packages/flutter_launcher_icons. Instancie e pacote no arquivo `pubspec.yam` do projeto na seção de dependências de desenvolvimento, deverá ficar assim:

```
dev_dependencies:
    flutter_launcher_icons: ^0.7.4
```

No mesmo nível de tabulação do `dev_dependencies` , insira

os dados a respeito do ícone que será utilizado para o aplicativo. É importante que você utilize uma imagem `.png` sem fundo em formato quadrado e de preferência com no mínimo 512 x 512 pixels de tamanho. O ideal é que seja 1024 x 1024 pixels. A declaração dos `assets` de ícone deverá ficar assim:

```
flutter_icons:
  image_path: "assets/icon.png"
  android: true
  iOS: true
```

Não se esqueça de criar na raiz do projeto uma pasta chamada `assets` e inserir o ícone com exatamente o mesmo nome especificado no `image_path`, se não, não vai funcionar. E após fazer isso, certifique-se de que as dependências foram baixadas. Caso você esteja utilizando um editor de texto mais esperto configurado com a extensão para Flutter, assim como costuma ser o VS Code, ele baixará automaticamente. Caso contrário, execute em seu terminal de comando a seguinte linha:

```
flutter pub get
```

E por último, mande gerar os ícones utilizando o seguinte comando:

```
flutter pub run flutter_launcher_icons:main
```

Como estamos aprendendo Flutter através de um livro, escolhi um ícone de uma estante de livros. Veja como ficou o nosso aplicativo:

Figura 13.2: Ícone personalizado no Flutter

Para ver o código deste projeto na íntegra acesse o seguinte endereço:

https://github.com/Leomhl/flutterbook_icone

Capítulo 14

SERÁ O FLUTTER A BALA DE PRATA?

14.1 A BALA DE PRATA

Caso não tenha visto o termo "bala de prata" inserido no meio da área de sistemas, esta será a primeira vez de muitas que você verá. O conceito de bala de prata nasceu em plena Era do misticismo da humanidade, quando as pessoas acreditavam que as balas de prata seriam o único tipo de munição que conseguia matar lobisomens, bruxas e monstros. Não apenas balas, flechas de prata também, de acordo com a crença, eram capazes de matar tais seres. O curioso é que por mais que houvesse o conceito de flecha, o que se perpetuou na história foi o termo "bala de prata".

Basicamente a bala de prata era o artefato que resolvia definitivamente um problema bem específico que normalmente gerava muita dor de cabeça para a população que acreditava cegamente em lobisomem, bruxa e afins. Antes das balas e flechas de prata, normalmente quando algum ser desses descritos anteriormente era "encontrado", havia luta corporal bastante próxima. Nem sempre o processo era simples e indolor e podia levar várias horas até que uma das partes sucumbisse. Em resumo, era de baixa eficiência, demorava bastante e tinha um elevado risco

envolvendo a vida do destemido herói que tentaria matar a besta fera. Com a invenção das armas e adaptação dos arcos de flecha para o disparo de balas de prata, o problema se tornou muito simples de resolver. Era só dar um bom disparo direcionado de uma distância segura que o problema em segundos era resolvido. Assim, a bala de prata tornou bastante rápido, seguro e direto o procedimento para resolver tamanho problema que havia na idade média. Ok, a história é legal mas o que tem a ver superstição da Idade Média com computadores, linguagens de programação e tecnologia? No conceito em si, até que bastante coisa.

Quando precisamos adotar uma linguagem de programação, framework, banco de dados ou qualquer outra ferramenta tecnológica que seja capaz de resolver nossos problemas complexos com grande eficiência e de maneira simples, podemos chamá-la de bala de prata. Basicamente a expressão foi adotada como uma metáfora que arremete a uma solução simples para um problema complexo. Resolver problemas é o nosso trabalho, e, após toda esta explicação a bala de prata se torna um termo que representa bastante a escolha sábia de tecnologias para uma solução criativa de problemas propostos.

14.2 O FLUTTER É UMA BALA DE PRATA?

Se você entrou neste capítulo acreditando ler algum tipo de resposta como "sim, é uma bala de prata" ou "não, não é uma bala de prata", sinto lhe dizer, mas, nenhuma das duas frases será dita aqui. A ideia deste capítulo como um todo é que você decida por conta própria se Flutter é ou não uma bala de prata para o projeto ou projetos em específico que você quer criar. Não existe linguagem genérica para todos os problemas, muito menos

frameworks. Seria um mundo completamente ideal caso isso ocorresse, mas não ocorre. Programação é um ramo bastante complexo que demanda foco e boas habilidades técnicas para selecionar as tecnologias ideais na hora da construção de um projeto e o programador TEM QUE extinguir o amor por uma tecnologia específica na hora das decisões e escolher a que melhor serve ao projeto, não a de que ele mais gosta.

O grande erro da maioria dos programadores é achar que qualquer linguagem ou framework serve bem a maioria dos projetos. Na esmagadora parte dos casos, até conseguimos fazer o típico "empurra que pega", mas não é porque chave de fenda e um pouco de jeitinho abre um parafuso formato philips que ela será a melhor ferramenta para esta tarefa no médio e longo prazo. Ser pragmático é fundamental na escolha de tecnologias, então, ao escolher Flutter, seja bastante pragmático. Por mais lindo e apaixonante que o Flutter possa ser, eu mesmo precisei mudar o framework mobile de um projeto em que trabalhei onde Flutter seria o escolhido. Qual era a razão disso? Algumas APIs de mapa não funcionavam da maneira como precisávamos e este não funcionamento inviabilizava a lógica de negócio do projeto. Em resumo, por mais fantástico que possa ser, o Flutter não atendia àquele projeto em específico. Isso quer dizer que ele é ruim? De maneira alguma! Só que ele não era a bala de prata certa para aquele monstro naquele momento. Nessas horas precisamos engolir o amor pelo framework e empregar muito profissionalismo e pesquisas para determinar o que usar.

Como encontrar a bala de prata?

Encontrar a bala de prata para um projeto não é uma tarefa

simples. Vários aspectos precisam ser levados em conta para que o sucesso da escolha da tecnologia seja obtido. É importante no início de um projeto validar quais são os itens essenciais para sustentar a lógica de negócio. Imagine só você ter como premissa fundamental para a existência do projeto e obtenção de investimentos financeiros que seu aplicativo capture dados de geolocalização com o aplicativo em segundo plano. E após meses de desenvolvimento quando a equipe finalmente terminou de fazer a interface gráfica do aplicativo, telas de autenticação, API, modelagem de banco de dados e afins, todo mundo descobre que o framework escolhido simplesmente não executa nada em background? Pois é... Já era o projeto. E infelizmente isso acontece com muito mais frequência do que gostaríamos. A "não falência" de uma empresa está muita das vezes nas mãos do desenvolvedor e ele próprio não tem noção disso. Para auxiliar com a escolha de um bom framework mobile vamos detalhar alguns pontos importantes dos três principais frameworks mais utilizados no mercado mobile brasileiro enquanto este livro é escrito, sendo eles: Ionic, React Native e Flutter.

14.3 IONIC + REACT NATIVE + FLUTTER

Antes de detalharmos a finalidade de cada um deles em específico, é interessante levantar os pontos em comum dos três. Todos eles trabalham com o conceito de *single code base* (única base de código). Assim, você não precisa escrever dois aplicativos, um para o Android e outro para o iOS. Todos estes frameworks com uma única base de códigos conseguem gerar aplicativos para as duas plataformas. Um detalhe bastante interessante e que é válido ressaltar é que todos estes também têm acesso às

funcionalidades nativas do sistema operacional como: câmera, vibrar, acelerômetro, gps... Através de plugins disponibilizados em partes pela comunidade de desenvolvedores e pelas criadoras dos frameworks os desenvolvedores ganham os mesmos poderes (ou quase) de acesso ao dispositivo que um aplicativo criado em linguagem nativa tem.

Tanto o Ionic quanto o React Native e o Flutter têm empresas gigantes e renomadas de mercado utilizando-os, trazendo maior credibilidade e força da comunidade em apostar nestas tecnologias para trabalhar no cotidiano. Mais um ponto bastante relevante são os criadores das tecnologias empregadas nestes frameworks. O Ionic, antes da versão 4, utilizava exclusivamente o Angular que está sob os cuidados da Google. Após a versão 4, os programadores podem escolher se desejam utilizar React, Vue ou Angular. Por baixo dos panos do Ionic tem o projeto Cordova, em que a Adobe e a Apache Foundation investiram bastante tempo para deixar robusto e seguro para o desenvolvimento.

O React Native tem como pai e mantenedor o Facebook que não mede esforços para evoluir a tecnologia e corrigir possíveis falhas junto à comunidade. E por último, mas não menos importante, temos o Flutter que está aos cuidados do Google. Particularmente, acho interessante o esforço e engajamento do Google para manter o mercado de desenvolvimento móvel híbrido aquecido e ao mesmo tempo acho estranho a Apple não tender para alguma linha do gênero. Enfim, cada empresa com suas lógicas... Voltando aos frameworks, todos os três são excelentes meios para criarmos aplicativos móveis, mas existem particularidades que os tornam "bala de prata" ou não. E veremos este detalhamento nos próximos tópicos.

Ionic

O Ionic é o framework com que normalmente a comunidade tem mais experiência de trabalho. Foram muitos anos sendo o mais utilizado do mercado, várias empresas mantiveram equipes criando e dando suporte a diversos aplicativos feitos com ele. Existe bastante respeito e carinho por esse framework, afinal, com certeza foi o primeiro framework móvel de muitos desenvolvedores que anteriormente trabalhavam com a web.

Apesar do carinho, precisamos ser profissionais. Reservo uma frase para ele, que sempre falo quando alguém me pergunta sobre o que eu acho. Basicamente a frase é: "Foi bom enquanto era bom". Não que ele seja ruim hoje, tanto o Angular quanto o core do Ionic em si evoluíram bastante nos últimos anos, mas a amarração até recentemente sem flexibilização ao Angular fez muita gente desistir dele e a comunidade perdeu força. Optar por ele ser um framework agnóstico foi uma ótima saída, mas perderam o *timing* de lançamento e o grande público já havia migrado para o React Native.

Outros pontos que são interessantes ressaltar além da redução da comunidade é a obsolescência de plugins. Muitos plugins extremamente úteis e críticos para a lógica de negócio das empresas estão sem receber atualização pelo criador há anos. Quer um exemplo? Um plugin que fez um projeto de que participei precisar mudar grande parte da lógica de negócio da aplicação foi o de pagamento dentro do aplicativo. Simplesmente não é confiável e dá mais problemas do que funciona. Se hoje perguntarem o que usar para um aplicativo que recebe pagamentos, nossa equipe não recomendaria o Ionic. Outro ponto

importante é que ele trabalha pessimamente com operações em background. Por mais que tenham criado plugins, modificado uma coisa daqui e dali, no fundo não passa de uma gigante gambiarra que consome mais bateria do que funciona. O código criado no Ionic não está muito próximo do código nativo. Ele é, em grosso modo, uma web view ("browser") embutido em um projeto nativo que renderiza seu código. Isso o torna bastante pesado dependendo do tipo que coisa que for rodar. Nem sempre o usuário gosta de esperar a renderização dos elementos na tela, o que pode acarretar em reclamações.

Agora vamos deixar as pedras nas mãos de lado e detalhar as bonanças do Ionic. Caso precise criar um projeto que se resuma a formulários e poucos acessos a recursos do dispositivo (câmera, touch id...) ele é excelente. Qualquer equipe que trabalha com desenvolvimento web (HTML, CSS, JS) consegue dentro de poucos dias criar um aplicativo que atenda a demanda. A curva de aprendizado dele também é ótima, de forma bastante rápida é possível aprender vindo do mundo web como lidar com telas e os componentes da aplicação.

Outra aplicabilidade muito legal do Ionic é para prototipação. Quer prototipar uma ideia de aplicativo mas não dispõe de muito tempo, dinheiro e conhecimentos fora de web? Vá para o Ionic. Caso o protótipo vingue e precise evoluir, é recomendável pesquisar bem profundamente se vale ou não continuar investindo nele. Mas para algo preliminar é excelente.

No geral, Ionic é bom para projetos menores e que já exista uma equipe com bons conhecimentos de web, assim, é possível criar algo que atenda todas as necessidades e economize recursos.

React Native

Criado pelo Facebook, o React Native veio com uma proposta melhorada se compararmos com a proposta do Ionic. Por mais que ele utilize JavaScript como linguagem principal, a organização dos componentes não é feita com HTML e CSS. Ele consegue converter com louvor os componentes escritos em JavaScript para componentes nativos. O formato da notação é conhecido como JSX, uma mistura de JavaScript com XML. Normalmente os desenvolvedores não acham o JSX bonito nem feio; a palavra que melhor tenho para descrevê-lo é "diferente". Então, o produto final é um aplicativo quase nativo. Existe uma ponte de comunicação entre algumas funcionalidades do JavaScript com o sistema operacional, mas, nos pormenores, ele apresenta uma performance muito superior quando comparamos com o Ionic.

Um ponto ruim é que, se você quiser aproveitar o código do aplicativo para criar uma plataforma web, não será possível realizar esta ação com a mesma simplicidade que seria possível fazer com o Ionic. Ele não é focado em web e, sim, em mobile. Apesar de utilizar uma linguagem web, a curva de aprendizado dele é um pouco maior que a do Ionic já que o primeiro contato com o React não é tão amistoso.

A comunidade de desenvolvedores é bem ativa e são os raros plugins que deixam na mão. Nenhum framework está livre de problemas com pacotes de terceiros, mas, pela maturidade e tempo de mercado do React Native, ele tem dado um bom show quanto a isso. Caso seu projeto não tenha como extrema premissa principal desempenho e você disponha de pessoas com conhecimento em JavaScript, é válido o investimento nele.

Flutter

Finalmente, o Flutter! Queridinho da maioria, auge do *hype* e uma promessa de chutarmos de vez as pontes de comunicação e viver de comunicação direta com o ambiente nativo. Muita coisa ainda precisa melhorar no Flutter. Apesar de o amarmos e investirmos horas e horas em pesquisas, ele ainda é uma escolha delicada para muitos projetos. Por ser bastante recente no mercado comparando com o Ionic e React Native, centenas de pacotes e plugins simplesmente ainda não foram portados de JavaScript para o Dart e isso pode gerar um problemão para o seu projeto.

A comunidade Flutter tem muitos simpatizantes que acabam criando um falso volume de "comunidade grande", mas, que tem poucos especialistas de verdade criando coisas novas e realmente úteis. Antes de começar a escrever código em Dart é importante escrever a lista de funcionalidades principais que seu aplicativo precisa ter e pesquisar. Pesquise muito se o Flutter implementa ou não estas funcionalidades e se existem pacotes atualizados de terceiros que façam o que você precisa. O Flutter é sim promissor, mas, ter pé no chão às vezes faz parte do processo! Cuidado na hora de escolher suas ferramentas.

É recomendado utilizar projetos com o Flutter quando performance for indispensável e não existir a possibilidade de criar de maneira nativa o aplicativo. É importante ressaltar que mão de obra verdadeiramente especializada em Dart ainda é bastante escassa e vai demorar um tempinho até que os desenvolvedores capacitem-se nela. Por ser uma tecnologia recente, o Flutter pode sofrer várias "avarias" do Google ao longo do processo que talvez exija refatoração de funcionalidades por parte dos

desenvolvedores. É essencial colocar todas essas coisas na balança.

De volta para a bala de prata

E agora, consegue enxergar melhor o que utilizar em seus próximos projetos? O Flutter tem sim se mostrado em uma boa direção e tende a ser uma excelente bala de prata para o mundo móvel. Combina performance com o JIT e AOT combinados que a linguagem Dart traz, tem uma comunidade crescente e principalmente uma gigante da tecnologia por trás visando garantir sua evolução. Ainda precisa de um aperto aqui e ali nos parafusos, mas nada que com o tempo e dedicação do Google e da comunidade não se resolva. Lembre-se, sabedoria é chave na escolha de tecnologias para o desenvolvimento em geral, não só de aplicativos.

14.4 E É AQUI QUE A SAGA COMEÇA!

Uau! Viajamos através de vários assuntos envolvendo a arquitetura da linguagem Dart, razões pelas quais o Flutter foi criado e está em alta no mercado assim como podemos criar aplicações bastante funcionais ao longo dos nossos dias. É praticamente impossível escrever em tempo hábil um livro completíssimo sobre o Flutter abordando tudo o que existe até porque todos os dias nascem novas bibliotecas e `Widgets` e morrem os anteriores. Caso fizesse, provavelmente tomaria alguns anos para a realização da escrita e quando publicássemos grande parte do framework já teria mudado.

A ideia deste livro foi introduzir a base, o norte magnético para você seguir adiante na caçada por conhecimento. Explore ao

máximo a documentação da linguagem Dart, do framework Flutter e tudo o que as comunidades de desenvolvedores publicam. A união para o crescimento e entendimento de uma tecnologia é crucial para todos nós. É através dessas tecnologias que conseguimos escrever uma vírgula na história da humanidade tornando algo melhor para outras pessoas.

Trabalhamos com tecnologia, temos este poder, utilize-o sem medo. Foi um enorme prazer compartilhar tantos conhecimentos legais e ter a sua atenção até aqui. Desejo muito sucesso em sua caminhada pelo mundo móvel e, assim como um grande artista, crie verdadeiras obras de arte com o Flutter!